Facultad de Matemática, Astronomía y Física
Universidad Nacional de Córdoba

Falluto: Un model checker para la verificación de sistemas tolerantes a fallas

Edgardo E. Hames
Director: Dr. Pedro R. D'Argenio

Córdoba, 14 de diciembre de 2009

A mi familia, porque este trabajo también es fruto de su esfuerzo.
A María Carolina, por su amor, dulzura y compañía.
A Julio, por tantos años de amistad cómplice.
A Pedro, por su infinita ayuda y paciencia.

Vuela nomás palomita,
El cielo te espera allá,
Se libre como los vientos
Yo ya te voy a alcanzar.
...
El que sueña es como un río
Que va buscando la mar
Se hace rumbo con el viento
Y llega a cualquier lugar.

Chacho Echenique, *Madurando sueños*

Resumen

Muchos equipos informáticos de nuestra vida cotidiana necesitan aplicaciones de alta disponibilidad (p. ej. equipamiento médico), y otros operan en lugares de difícil acceso (p. ej. equipos en el espacio). Estos sistemas están sujetos a una variedad de fallas potenciales que pueden corromper o degradar su rendimiento. La tolerancia a fallas no se logra simplemente por redundancia de módulos y es necesario aplicar técnicas sistemáticas para establecer que funcionan correctamente.

El model checking es una técnica de verificación formal que, dado un modelo de un sistema y una propiedad requerida, permite decidir automáticamente si la propiedad es satisfecha o no. A diferencia del testing y de la simulación, el model checking es capaz de evaluar todos los escenarios posibles de un sistema y determinar su corrección.

Este trabajo presenta *1)* un lenguaje declarativo para el modelado de fallas, *2)* FALLUTO, una herramienta de model checking para la verificación de sistemas tolerantes a fallas, *3)* un método de verificación para dichos sistemas y *4)* su aplicación a dos casos de estudios reales en sistemas distribuidos, haciendo uso del lenguaje y la herramienta.

Clasificación: D.2.4 Software / Program Verification

Palabras clave: ingeniería de software, verificación de sistemas, model checking, tolerancia a fallas

Abstract

Many computer systems in our daily life require high availability applications (for example, medical equipment) and some others run on difficult-to-access places (for example, satellites). These systems are subject to a variety of potential failures that may jeopardize their performance. Fault tolerance cannot be achieved just by adding redundant modules to a system, and systematic techniques need be used to determine its correct behavior.

Model checking is a formal technique that can automatically decide whether the given model of a system satisfies a required property. In contrast with testing and simulation, model checking explores all possible states and behaviors of a system.

This work presents *1)* a declarative language for fault modelling, *2)* FALLUTO, a model checking tool for the verification of fault tolerant systems, *3)* a method for verifying those systems and *4)* its application on two case studies of real distributed systems, using the language and the tool.

Classification: D.2.4 Software / Program Verification

Keywords: software engineering, program verification, model checking, fault tolerance

Agradecimientos

Muchas personas han colaborado directa e indirectamente en la concreción de este trabajo. Nombro sólo a algunos que en este momento vienen a mi mente, sin un orden particular.

A Carlitos y Cecilia, que revisaron muchísimas veces este texto y corrigieron horrores gramaticales que escapan al ojo inexperto y que mejoraron sustancialmente este trabajo.

A mis padres, por soportar tantas ausencias durante el tiempo que llevó este trabajo.

A María Carolina, por aguantar la desazón que me acompañó en muchas ocasiones.

A Julio, que me acompañó en la ejecución de muchos proyectos de la carrera y en $|\mathbb{N}|$ horas de estudio.

A Pedro D'Argenio, que me propuso la lista de temas de trabajo final más larga que cualquier director pueda ofrecer, aceptó mis "decisiones de diseño", repasó conmigo cada tema, corrigió y revisó este trabajo sin aflojar.

A Daniel Gutson, por sus comentarios sobre parsers, árboles y patrones de diseño.

A Fabio Bustos, por autorizarme tantas faltas al trabajo para que pudiera estudiar y rendir las últimas materias y preguntarme constantemente "Flaco, ¿cuánto te falta para recibirte?".

Finalmente, a mis amigos de la Picada Moreana, por tantas horas de alegría y tantos partidos de fútbol.

Índice general

Capítulo 1

Introducción

En nuestra vida cotidiana, es fácil descubrir la ubicuidad de equipos informáticos. Algunos de ellos, necesitan aplicaciones de alta disponibilidad (p. ej. equipamiento médico, medios de transporte) y otros operan en lugares de difícil o imposible acceso (p. ej. equipo en el espacio o en ambientes peligrosos). Estos sistemas están sujetos a una variedad de *fallas* potenciales que pueden corromper o degradar su rendimiento. Mientras mayor sea la complejidad del sistema computacional o la duración de la ejecución de un algoritmo, mayor es el riesgo de aparición de una falla y de que ésta vuelva inservible el resultado [22].

En algunos casos, la confiabilidad de un sistema puede incrementarse con componentes críticos redundantes, es decir, con múltiples instancias de éstos. De este modo, es necesario que todos los componentes fallen para que el sistema en su conjunto falle. Sin embargo, la incorporación de tolerancia a fallas no se logra simplemente por mera redundancia de módulos. Un ejemplo paradigmático de un sistema cuyo módulo de tolerancia a fallas no dio resultado ocurrió en el primer lanzamiento del Ariane 5, el cual tuvo un desenlace no exitoso [27]. Por lo tanto, es deseable construir sistemas que puedan tolerar la ocurrencia de fallas internas (p. ej. división por cero) o externas (p. ej. la pérdida de un mensaje en una red de comunicación).

Idealmente, los algoritmos son derivados por un proceso sistemático conducido por argumentos formales que aseguran su corrección respecto a la especificación original [5]. En cambio, en la realidad contemporánea, los diseñadores suelen tener un argumento informal en mente y desarrollan el algoritmo final y sus parámetros explorando variaciones locales contra este argumento y contra escenarios que resalten casos difíciles o problemáticos [37].

La exploración de estos escenarios puede realizarse usando simuladores o prototipos que incrementen el número de escenarios que pueden ser examinados y la confiabilidad del resultado. Más aún, esta exploración puede ser

exhaustiva utilizando técnicas de *model checking* [37] que permitan verificar formalmente propiedades del sistema.

Model checking es una técnica de verificación que, dado el modelo del sistema bajo estudio y la propiedad requerida, permite decidir automáticamente si la propiedad es satisfecha o no. La principal desventaja del model checking es la *explosión de estados* que puede ocurrir cuando el sistema bajo verificación tiene muchas componentes con potenciales transiciones en paralelo. En este caso, el número de estados puede crecer exponencialmente con el número de procesos.

En este trabajo, definimos un lenguaje declarativo para el modelado de fallas y FALLUTO, una herramienta para la verificación de *sistemas tolerantes a fallas* mediante técnicas de model checking. FALLUTO brinda una capa alrededor de NuSMV, un model checker basado en diagramas de decisión binaria. Esta capa se implementa por medio de tres herramientas: un compilador que convierte el código de FALLUTO a código de NuSMV, un intérprete de trazas que "limpia" los contraejemplos de las especificaciones para que sólo contengan variables del modelo original, y un visualizador de trazas que produce un autómata a partir del contraejemplo limpio para ayudar a la comprensión del problema. Además, damos un proceso de verificación para sistemas con fallas y mostramos su aplicación en dos casos de estudio reales.

Otros autores ya han enfrentado el problema de la verificación de sistemas tolerantes a fallas. En 2001, Yokogawa et al. [YokogawaTK01] desarrollaron un lenguaje de modelado y un método para traducir los modelos a SMV usando lógica CTL. En 2007, Bozzano et al. [6] implementaron una extensión de NuSMV para el modelado e inyección de fallas. Lamentablemente, el uso y la distribución de la herramienta están restringidos a los socios patrocinadores del desarrollo. Más recientemente, en 2008, Minamikawa et al. [31] proponen otro lenguaje para modelar sistemas distribuidos y una herramienta para la compilación automática a PROMELA, el lenguaje del model checker SPIN.

FALLUTO y el proceso de model checking propuestos en este trabajo presentan algunas diferencias con respecto a investigaciones anteriores que lo hacen interesante. En primer lugar, se provee un lenguaje declarativo (no procedural) para las fallas. Por lo tanto, el usuario no tiene que modelar detalles tales sobre la ocurrencia de una falla o la restauración del sistema. Además, con la metodología propuesta, se puede reusar el modelo original del sistema por verificar y extenderlo para representar sus fallas. Por último, la herramienta es software libre y puede ser distribuida, estudiada y extendida por otros desarrolladores más allá del autor original.

Este informe está organizado de la siguiente manera: el capítulo 2 presenta las nociones básicas de programas y fallas que usaremos a lo largo del trabajo. El capítulo 3 presenta la lógica temporal lineal, un formalismo que

nos permite razonar sobre fórmulas y su relación en el tiempo. El capítulo 4 introduce el model checking, sus problemas y un proceso para aplicarlo a sistemas reales. En este capítulo, también presentamos NuSMV, el motor de verificación usado por nuestra herramienta. Los capítulos 5 y 6 están fuertemente vinculados entre sí, ya que presentan la sintaxis del lenguaje de FALLUTO y cómo se convierte a código de NuSMV. También se presentan algunas ventajas de la herramienta y sus limitaciones. El capítulo 7 vuelve a los conceptos teóricos y cataloga modelos de fallas que se encuentran en los sistemas reales y su implementación en FALLUTO. El capítulo 8 muestra la aplicación de la herramienta y el proceso de verificación en dos sistemas reales concretos. El capítulo final resume los aspectos principales de este trabajo y se presentan algunos posibles trabajos futuros. En los apéndices, presentamos la sintaxis completa del lenguaje de FALLUTO y el manual de uso de de la herramienta. Además, se incluye material complementario al texto principal del trabajo de grado.

Capítulo 2

Tolerancia a fallas

> ¿Qué es la tolerancia? Es la panacea de la humanidad. Todos los hombres estamos llenos de debilidad y de errores, y debemos perdonarnos recíprocamente pues ésta es la primera ley de la Naturaleza.
>
> Voltaire

La confiabilidad suele ser presentada como uno de los principales beneficios de los sistemas distribuidos. Normalmente, dicha fiabilidad está definida en términos estadísticos: p. ej. estableciendo la probabilidad de que el sistema esté operativo en un momento dado. Mientras que algunos términos tales como la "confiabilidad" y "disponibilidad" son importantes en entornos prácticos, no son centrales a este trabajo ya que nos enfocamos en la fase de diseño de un sistema y no en la de evaluación. Si bien los términos anteriores quedarán un poco difusos, es importante definir con precisión las características de un sistema distribuído. En este capítulo, veremos primero qué entendemos por programa y falla, y luego qué significa que un programa tolere un conjunto de fallas [2, 19, 28, 1, 3].

2.1. Programa

Podemos ver un *programa* como un conjunto de variables y un conjunto finito de procesos. Cada variable tiene un dominio predefinido no vacío y cada proceso consiste de un conjunto finito de acciones de nombre único de la forma:

$$nombre :: \langle guarda \rangle \rightarrow \langle comando \rangle$$

donde la *guarda* es una expresión booleana sobre las variables del programa y *comando* cambia el valor de cero o más variables y siempre termina.

Dado un programa p, diremos que el valor de todas las variables de p forman el *estado de p*. Una acción estará *habilitada* en un estado si y sólo si su guarda se hace verdadera en ese estado. Un proceso estará *habilitado* en un estado si y sólo si alguna acción del proceso está habilitada en ese estado.

2.2. Especificación

Una *propiedad* de un programa es un conjunto de ejecuciones. Diremos que un programa p *satisface* una propiedad ϕ si el conjunto de ejecuciones de p desde una configuración inicial es subconjunto de ϕ.

Distinguimos dos tipos de propiedades llamadas *safety* y *liveness*. Intuitivamente, una propiedad de *safety* nos dice que nada "malo" ocurre. Por ejemplo, "el sistema nunca llega a una situación de *deadlock*". Formalmente, es un conjunto S tal que para toda ejecución σ que no pertenece a S, existe un prefijo α de σ tal que para toda ejecución β ocurre que $\alpha\beta$ no pertenece a S. Esto nos dice que una propiedad de safety elimina los prefijos no deseados de una ejecución.

Nótese que un proceso que no realiza ninguna acción satisface trivialmente la propiedad de safety. Sin embargo, también queremos poder establecer el progreso de los procesos. La propiedad de *liveness* nos asegura que algo "bueno" ocurrirá en algún momento. Por ejemplo, "el programa termina". Formalmente, es un conjunto S tal que para cada ejecución parcial α en S, existe una ejecución β tal que $\alpha\beta$ pertenece a S. Esto nos dice que una propiedad de liveness elimina los sufijos no deseados de una ejecución.

Una *especificación de un problema* consta de una propiedad de safety y una propiedad de liveness. Un programa se dice *correcto* respecto a una especificación si y sólo si satisface estas propiedades de safety y liveness.

2.3. Falla

El Glosario de Términos de ATIS [15] define una *falla* como la condición accidental que puede provocar que una unidad funcional *fracase*[1] al realizar la función requerida.

Nosotros modelaremos una falla como una acción que cuando se ejecuta perturba el estado del sistema. Por lo tanto, una falla será una transición

[1]Dado que los vocablos *fault* y *failure* del inglés se traducen al español como falla, los distinguiremos utilizando falla para referirnos a fault y fracaso o avería para failure.

no deseada pero aun así posible. Naturalmente, la ocurrencia de una falla producirá un cambio en el comportamiento del sistema; un modelo de falla describirá cómo los componentes del sistema manifiestan el comportamiento fallido. Dado que modelamos una falla como una acción de programa, podemos ver un modelo de fallas como el agregado de comportamiento a un programa.

Formalmente, una *falla* es la ocurrencia de una acción virtual adicional del programa. Un *modelo de fallas* es una transformación de programa que mapea el programa p en p'. El program original p se ejecuta en un entorno ideal sin fallas; p' es el programa p sujeto a fallas. Nótese que el programa p' nunca será implementado explícitamente y el proceso de transformación simplemente es una ayuda para razonar sobre programas sujetos a fallas.

2.4. Tolerancia a fallas

Las transiciones de un sistema sin fallas siempre se dan dentro de un conjunto cerrado de estados: los estados "legales" del sistema (plano inferior en la Figura 2.1). Llamemos S al predicado que identifica ese conjunto de estados legales. Sea c un estado cualquiera del programa en el que se satisface S, y F, un conjunto de fallas. Diremos que el sistema es *tolerante a* F si en el estado c, ninguna acción $f \in F$ invalida S. En este caso, el sistema *enmascara* a F.

Supongamos que, al producirse una falla, el sistema abandona el conjunto de estados donde se cumple S (pasa al plano superior en la Figura 2.1). Las transiciones se dan entonces entre un conjunto de estados "riesgosos". Si tras un número finito de acciones $f \notin F$, el sistema converge al conjunto de estados legales, también diremos que el sistema es tolerante a F (vuelve al plano inferior en la Figura 2.1). Caso contrario, el sistema no es tolerante a F (p. ej. la acción que sale fuera de la línea de puntos en el plano superior de la Figura 2.1).

Dados un modelo de fallas F y un programa p correcto respecto a la especificación ϕ, si el programa $F(p)$ es correcto respecto a ϕ, diremos entonces que p es *tolerante* a F respecto a ϕ.

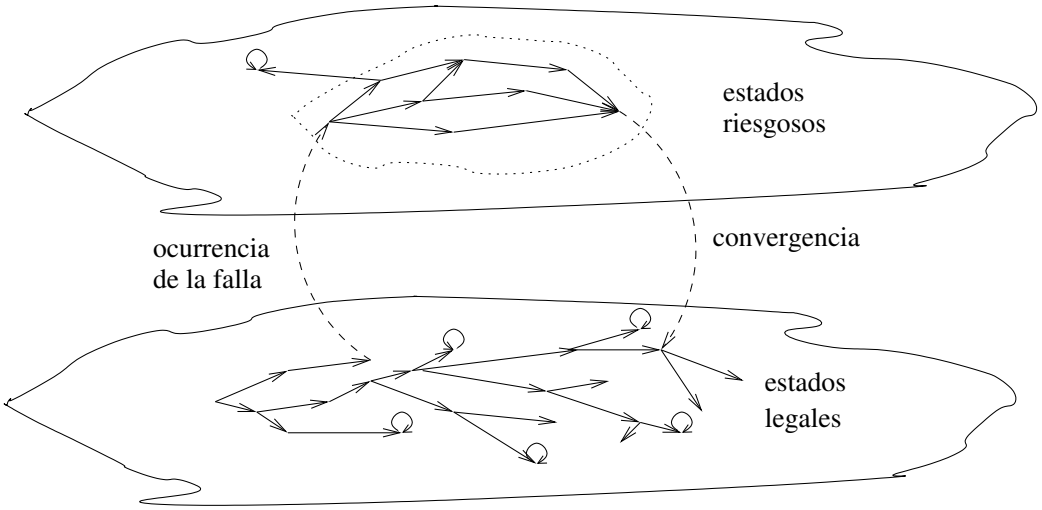

Figura 2.1: Transiciones de un sistema con fallas

Capítulo 3

Lógica Temporal Lineal

> ... no creía en un tiempo uniforme, absoluto. Creía en infinitas series de tiempos, en una red creciente y vertiginosa de tiempos divergentes, convergentes y paralelos. Esa trama de tiempos que se aproximan, se bifurcan, se cortan o que secularmente se ignoran, abarca todas las posibilidades. No existimos en la mayoría de esos tiempos; en algunos existe usted y no yo; en otros, yo, no usted; en otros, los dos.
>
> Jorge Luis Borges, *El jardín de los senderos que se bifurcan*

La lógica proposicional nos permite determinar el valor de verdad de una sentencia. Consideremos las siguientes proposiciones:

"Sócrates es mortal".

"El caballo blanco de San Martín es negro".

Habitualmente, diremos que la primera sentencia es verdadera y la segunda es falsa. Esto es así porque la lógica proposicional asocia un valor a una sentencia dentro de un único "mundo". Consideremos ahora la siguiente sentencia:

"Tengo sueño".

El valor de verdad de dicha proposición cambia con el tiempo: si estoy en una clase aburrida, será verdadera; por el contrario, si recién me levanto de la siesta, será falsa. Descubrimos entonces que la lógica proposicional es insuficiente para reflejar el valor de verdad de aquellas sentencias en las cuales el tiempo es relevante.

3.1. Lógicas modales

Las lógicas modales son un tipo de lógica que permite hacer consideraciones en un "conjunto de mundos". En nuestro caso estudiaremos las lógicas temporales, en las cuales los mundos son distintos momentos en el tiempo. De este modo, podremos hablar del valor de verdad de sentencias más interesantes:

"*Siempre* que llovió, paró".

"*En algún momento* terminaré de escribir este informe".

"Tendré sueño *hasta* que duerma".

En el contexto de la verificación de sistemas, estas lógicas son de suma importancia ya que nos permiten especificar propiedades tales como "los procesos nunca llegarán a un deadlock" o "todos los procesos tendrán una oportunidad de ingresar a su sección crítica".

Notemos que podemos tener distintas perspectivas del tiempo:

- Los instantes de tiempo forman un conjunto numerable.
- Los instantes de tiempo forman un conjunto denso.
- El tiempo evoluciona linealmente, y un instante se sucede a otro.
- El tiempo se va ramificando, y en cada instante se considera la alternativas de varios sucesos posibles que llevan a distintos estados.

En este trabajo, nos limitamos al estudio de la lógica temporal lineal (*Linear Temporal Logic*) [33, 4, 7], un formalismo que permite razonar sobre la ejecución de un programa con una perspectiva lineal y numerable del tiempo.

3.2. Lógica Temporal Lineal

LTL permite expresar el orden temporal de los cambios de estados de un sistema donde cada transición corresponde al avance de una unidad de tiempo. El momento presente hace referencia al estado actual y el momento siguiente, al estado inmediatamente posterior. Por lo tanto, el modelo del tiempo es discreto y el sistema es observable en cada uno de los instantes.

LTL provee los operadores proposicionales que ya conocemos ($\neg$, $\wedge$, $\vee$, $\rightarrow$) y preserva su semántica usual. Además, incorpora operadores modales, de los cuales daremos primero una noción intuitiva:

$\mathbf{X}\phi$ (del inglés, ne**X**t) "en el próximo instante vale ϕ"

$\mathbf{F}\phi$ (del inglés, **F**inally) "en algún instante futuro vale ϕ"

$\mathbf{G}\phi$ (del inglés, **G**lobally) "siempre en el futuro vale ϕ"

$\phi\mathbf{U}\psi$ (del inglés, **U**ntil) "vale ϕ continuamente hasta que vale ψ"

$\phi\mathbf{R}\psi$ (del inglés, **R**elease) "deja de suceder ϕ cuando sucede ψ"

Con esta descripción informal de los operadores modales, podemos escribir las propiedades de *safety* y *liveness* de la siguiente manera:

$$\begin{aligned} \text{safety} &= \mathbf{G}\neg \text{ "ocurre algo malo"} \\ \text{liveness} &= \mathbf{F} \text{ "ocurre algo bueno"} \end{aligned}$$

Veamos dos problemas clásicos: la exclusión mutua y los filósofos comensales.

Ejemplo 1 (Problema de exclusión mutua)**.** Sean P_1 y P_2 dos procesos concurrentes. Podemos modelar P_i con tres fases: (1) su sección no crítica, (2) la espera para entrar en la sección crítica (denotada $espera_i$), y (3) la sección crítica (denotada $crit_i$).

La propiedad de *safety* dice que P_1 y P_2 nunca están en sus respectivas secciones críticas simultáneamente. En este caso, la idea de "algo malo" es $crit_1 \wedge crit_2$; por lo tanto, la propiedad de *safety* expresada en LTL es

$$\mathbf{G}\neg(crit_1 \wedge crit_2)$$

La propiedad de *liveness* dice que cada proceso que esté esperando para entrar en su sección crítica lo conseguirá en algún momento:

$$(\mathbf{GF}espera_1 \rightarrow \mathbf{GF}crit_1) \wedge (\mathbf{GF}espera_2 \rightarrow \mathbf{GF}crit_2)$$

Se puede demostrar que $\mathbf{GF}p \rightarrow \mathbf{GF}q$ es equivalente a $\mathbf{F}(\mathbf{GF}p \rightarrow \mathbf{GF}q)$ y conseguir que la propiedad de liveness tenga la forma **F** "algo bueno ocurre".

Ejemplo 2 (Los filósofos comensales)**.** Cinco filósofos están sentados en una mesa redonda con un tazón con arroz en el centro. Para poder comer, necesitan dos palitos, y entre dos filósofos vecinos sólo hay uno. De tanto pensar, les da hambre; por lo tanto, alternan momentos de hambre con momentos de profundas elaboraciones. Una propiedad que debe satisfacer este sistema es que nunca deben estar todos los filósofos sosteniendo un palito y esperando que el vecino lo suelte. Es decir, que el sistema no se encuentra en un deadlock.

Sean $espera_i$ la proposición que dice "el filósofo i-ésimo sostiene un palito en la mano y espera que alguno de sus vecinos libere el otro" y $usado_i$ la proposición "el palito i-ésimo está siendo usado". La propiedad de safety es

$$\mathbf{G}\neg(\bigwedge_{0\leq i<5} espera_i \wedge \bigwedge_{0\leq i<5} usado_i)$$

Además, querremos que todos los filósofos puedan comer en algún momento. Sea $come_i$ la propiedad que dice "el filósofo i come", entonces la propiedad de liveness para los filósofos comensales es

$$\bigwedge_{0\leq i<5} \mathbf{GF}\, come_i$$

3.2.1. Sintaxis y semántica de LTL

Sea $\mathcal{PA}$ el conjunto de proposiciones atómicas. Cualquier proposición $p \in \mathcal{PA}$ es una fórmula LTL. Si ϕ y ψ son fórmulas LTL, todas las fórmulas construidas con los operadores lógicos

$$\neg\phi \qquad \psi \wedge \phi \qquad \psi \vee \phi \qquad \psi \Rightarrow \phi \qquad \ldots$$

son fórmulas LTL y también los son las construidas con los operadores temporales

$$\mathbf{X}\phi \qquad \mathbf{G}\phi \qquad \mathbf{F}\phi \qquad \psi\mathbf{U}\phi \qquad \psi\mathbf{R}\phi \qquad \ldots$$

Llamamos *traza* a cada uno de los elementos de $(2^{\mathcal{PA}})^\omega$. Sea $\sigma = A_0A_1\ldots$ una traza, el sufijo i-ésimo de σ es $\sigma[i..] = A_iA_{i+1}\ldots$. Denotamos con $\sigma \models \phi$ que la fórmula LTL ϕ se satisface en la traza σ, y la relación $\models$ se define inductivamente de la siguiente manera:

$\sigma \models true$		
$\sigma \models p$	sii	$p \in \sigma(0)$, para todo $p \in \mathcal{PA}$
$\sigma \models \neg\phi$	sii	$\sigma \not\models \phi$
$\sigma \models \phi \wedge \psi$	sii	$\sigma \models \phi$ y $\sigma \models \psi$
$\sigma \models \phi \vee \psi$	sii	$\sigma \models \phi$ o $\sigma \models \psi$
$\sigma \models \mathbf{X}\phi$	sii	$\sigma[1..] \models \phi$
$\sigma \models \mathbf{F}\phi$	sii	$\exists j \geq 0 : \sigma[j..] \models \phi$
$\sigma \models \mathbf{G}\phi$	sii	$\forall j \geq 0 : \sigma[j..] \models \phi$
$\sigma \models \phi\mathbf{U}\psi$	sii	$\exists j \geq 0 : \sigma[j..] \models \psi$ y $\forall i : 0 \leq i < j : \sigma[i..] \models \phi$
$\sigma \models \phi\mathbf{R}\psi$	sii	$\forall j \geq 0, \forall i < j : (\sigma[i..] \not\models \phi \Rightarrow \sigma[j..] \models \psi)$

Con esta definición de $\models$, podemos definir el lenguaje de ϕ como:

$$\mathcal{L}(\phi) = \{\sigma \in (2^{\mathcal{PA}})^\omega \mid \sigma \models \phi\}$$

Ya sabemos que los operadores proposicionales básicos son $\neg$ y $\wedge$, y que los demás pueden derivarse de éstos. En LTL, todos los operadores modales se derivan de **X** y **U**:

$$\begin{aligned} \mathbf{F}\phi &= true\mathbf{U}\phi \\ \mathbf{G}\phi &= \neg\mathbf{F}\neg\phi \\ \phi\mathbf{R}\psi &= \neg(\neg\phi\mathbf{U}\neg\psi) \end{aligned}$$

3.3. Equidad (*Fairness*)

Dado que varias acciones de un programa pueden estar habilitadas en el mismo momento, podríamos querer considerar sólo aquellas ejecuciones en las cuales el scheduler no ignora una de las acciones por siempre. Es decir, podemos restringir las ejecuciones que sean equitativas con respecto a las acciones. A estas restricciones, las llamaremos *restricciones de equidad*.

Sean las siguientes fórmulas lógicas ϕ "una acción está habilitada" y ψ "una acción es elegida", definimos las siguientes restricciones de equidad[1]:

Unconditional Fairness (equidad incondicional): Una acción es elegida infinitamente a menudo.

$$ufair = \mathbf{GF}\psi$$

Strong Fairness (equidad fuerte): Si una acción está habilitada infinitamente a menudo (no necesariamente siempre), entonces será elegida infinitamente a menudo.

$$sfair = \mathbf{GF}\phi \rightarrow \mathbf{GF}\psi$$

Weak Fairness (equidad débil): Si una acción está habilitada continuamente (no es inhabilitada temporalmente), entonces será elegida infinitamente a menudo.

$$wfair = \mathbf{FG}\phi \rightarrow \mathbf{GF}\psi$$

[1] La noción de fairness puede generalizarse más allá del caso particular presentado aquí. Se puede pensar ϕ como la condición de equidad y ψ como el efecto de la equidad.

Cabe notar la siguiente relación entre las restricciones de equidad:

$$ufair \Rightarrow sfair \Rightarrow wfair$$

Intuitivamente, podemos pensar el problema de la equidad de la siguiente manera. En un restaurant, desearemos que el mozo sea equitativo con los clientes, es decir, que no ignore permanentemente a los comensales de una mesa. El mozo podría optar por varios criterios para atender a todos los comensales:

- Un mozo con equidad incondicional pasa por cada mesa sin cesar y pregunta a los comensales si desean algo.
- Un mozo con equidad fuerte espera que los comensales levanten la mano para llamar su atención. Si los comensales se cansan, pueden bajar la mano, pero el mozo les garantiza que después de haber levantado la mano un número finito de veces, serán atendidos.
- Un mozo con equidad débil espera que los comensales levanten la mano para llamar su atención, pero no pueden bajarla. El mozo les garantiza que después de haber sostenido la mano en alto por un período finito de tiempo, serán atendidos.

Podemos ver que en el problema de la exclusión mutua presentado anteriormente, los procesos tienen garantizada la propiedad de liveness con la equidad fuerte y en el caso de los filósofos comensales, con la equidad incondicional.

3.4. Observaciones finales

En este capítulo sólo consideramos la perspectiva lineal del tiempo. Otra posibilidad es la del tiempo ramificado y es estudiada por la lógica CTL (*Computation Tree Logic*) [33]. No se puede comparar la expresividad de LTL con la de CTL, ya que hay sentencias en una que no pueden ser expresadas en la otra. Encontramos un ejemplo de diferencia de expresividad en las restricciones de equidad. Estas fórmulas no pueden expresarse en CTL. Sin embargo, nosotros las necesitaremos para permitir que el comportamiento de los procesos y la ocurrencia de las fallas se alternen equitativamente.

Capítulo 4

Model Checking y NuSMV

En un proyecto de desarrollo de software, la validación temprana del diseño es un desafío que ocupa un período significativo de tiempo. En muchas organizaciones, las prácticas más comunes para atacar este problema son la *simulación* y el *testing*. Si bien ambas técnicas comparten el objetivo de detectar errores, éstas difieren en el enfoque del problema. La simulación permite crear modelos que ayudan a comprender mejor los procesos, la arquitectura o el flujo de una operación antes de invertir tiempo y dinero en el desarrollo de un sistema real. Por otro lado, el testing permite ejecutar el sistema en distintos escenarios y validar si los resultados son correctos o si el sistema satisface los requerimientos técnicos y del dominio. En fases tempranas, cuando el diseño está inmaduro y repleto de *bugs*, estas técnicas resultan altamente efectivas para depurarlo [7]. Sin embargo, a medida que el diseño se va "limpiando" y refinando, la efectividad de estas técnicas comienza a decaer drásticamente. De este modo, es posible que los escenarios no explorados contengan bugs fatales que escapan a las pruebas y se hacen visibles en entornos de producción. El testing sólo puede probar la presencia de bugs pero no su ausencia [11].

Una alternativa atractiva es el uso de técnicas de verificación formal que permitan determinar si el sistema es correcto y asegurar que todos los comportamientos posibles del sistema hayan sido probados. El *model checking* es una técnica automática de verificación de sistemas concurrentes finitos. Dados un modelo M de un sistema y una propiedad ψ, un model checker es capaz de establecer si M satisface ψ y, en caso de no satisfacerla, presenta una ejecución que la hace falsa. Como se puede notar, a diferencia de la simulación y del testing, el model checking apunta a establecer la corrección del sistema.

El model checking [7] presenta ventajas frente a otras alternativas:

- Su aplicación permite automatizar el proceso de verificación y no requiere supervisión ni experiencia en ramas de la matemática, como la lógica o la demostración de teoremas. Cualquier usuario capaz de ejecutar simulaciones de un diseño puede verificarlo con un model checker.
- Cuando el diseño no satisface la propiedad, el model checker presenta un contraejemplo que demuestra un comportamiento que hace falsa dicha propiedad.

El principal problema del model checking es la *explosión del espacio de estados* y existen varias técnicas para manejarlo. Por ejemplo, la *reducción de orden parcial* define una relación de equivalencia entre comportamientos indistinguibles para la propiedad y, en lugar de construir el grafo de estados completo, construye uno reducido que sólo contiene representantes de las distintas clases de equivalencias de las ejecuciones. Otra posibilidad es utilizar *diagramas de decisión binaria* (*Binary Decision Diagrams*, *BDD*) que representan funciones booleanas de un modo muy compacto y permiten manipular grandes cantidades de estados.

Existen muchos casos notables de uso de model checking en sistemas reales [8], por ejemplo:

IEEE Futurebus+. Verificación del protocolo de coherencia de cache del Standard 896.1 1991.

ISDN/IUPP Verificación del desarrollo del protocolo ISDN User Part Protocol en AT&T.

PowerScale. Verificación del protocolo de arbitraje de bus de la arquitectura multiprocesador usada en los servidores y estaciones de trabajo Bull Escala.

Edificios. Verificación del sistema de control estructural activo para compensar movimientos sísmicos en la Universidad Estatal de Carolina del Norte.

4.1. Proceso de Model Checking

Antes de validar el diseño con un model checker, el usuario debe preparar [34]:

- el modelo que se debe verificar, y
- el conjunto de propiedades que el sistema debe satisfacer.

Podemos así distinguir los pasos del proceso de model checking [7], ilustrado en la Figura 4.1.

Modelado. El primer paso es convertir el diseño al lenguaje del model checker. A veces, el diseño tiene demasiados detalles y es necesario abstraerlo para contener la explosión de estados.

Formalización. Este paso involucra detallar las propiedades que el diseño ha de satisfacer. Las propiedades deben estar dadas en alguna formula lógica que el model checker comprenda. Es frecuente el uso de la *lógica temporal* para indicar cómo se espera que evolucione el sistema con el paso del tiempo.

Verificación. Idealmente, este paso es automático. Sin embargo, en la práctica se involucra al usuario, quien debe analizar los resultados de la verificación, hacer modificaciones al modelo, los requerimientos o la especificación, y en algunos casos también debe ajustar los parámetros del model checker.

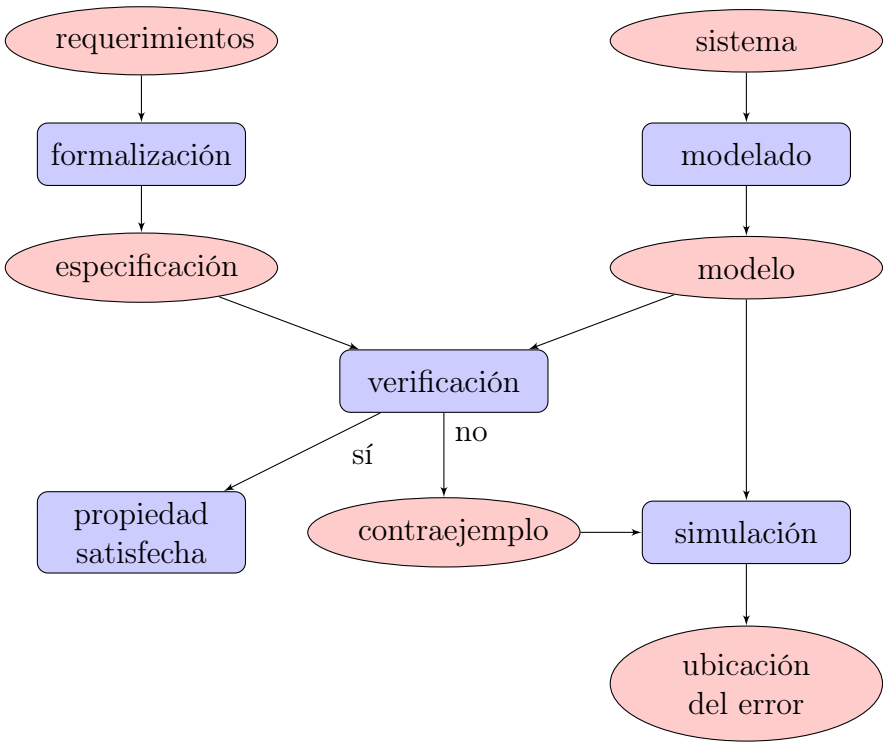

Figura 4.1: Proceso clásico de model checking

4.1.1. Model checking y tolerancia a fallas

La metodología propuesta en este trabajo para la verificación de sistemas tolerantes a fallas incorpora una etapa más de verificación (ver figura 4.2):

1. Obtener un modelo M del sistema.
2. Formalizar los requerimientos en una especificación ψ.
3. Verificar que el sistema satisfaga la propiedad deseada ($M \models \psi$).
4. Para cada falla f a la que el sistema está expuesto, aplicar el modelo de f a M.
5. Debilitar la especificación original para contemplar los requerimientos de fallas y obtener ψ'.
6. Verificar que el modelo del sistema M' satisfaga la especificación ψ' ($M' \models \psi'$).

En algunos casos, la propiedad que queremos verificar en el último paso no es exactamente ψ. Supongamos que queremos verificar que el sistema M satisface la propiedad de safety ψ. En un sistema sin fallas, la especificación será "siempre se cumple ψ". Consideremos ahora que el sistema está sujeto a fallas y que implementa redundancia triple. En este caso, la nueva especificación (ψ') será "siempre que ocurran menos de tres fallas, se cumple ψ".

4.2. NuSMV

NuSMV [14] es un model checker simbólico (utiliza fórmulas booleanas para representar estados y transiciones) originado en la reingeniería, reimplementación y extensión de SMV [30], el model checker simbólico basado en BDD.

Funcionalidad. Permite repesentar sistemas finitos síncronos y asíncronos, y el análisis de especificaciones expresadas en *Computational Tree Logic* (CTL) y *Linear Temporal Logic* LTL mediante técnicas de model checking basadas en BDD y SAT.

Arquitectura. El sistema está dividido en varios módulos con interfaces claras que proveen *parsing* del modelo, la conversión del modelo escalar en un modelo booleano, la construcción del modelo en BDD, la verificación de BDD, el model checking acotado, y la simulación y manipulación de trazas [13].

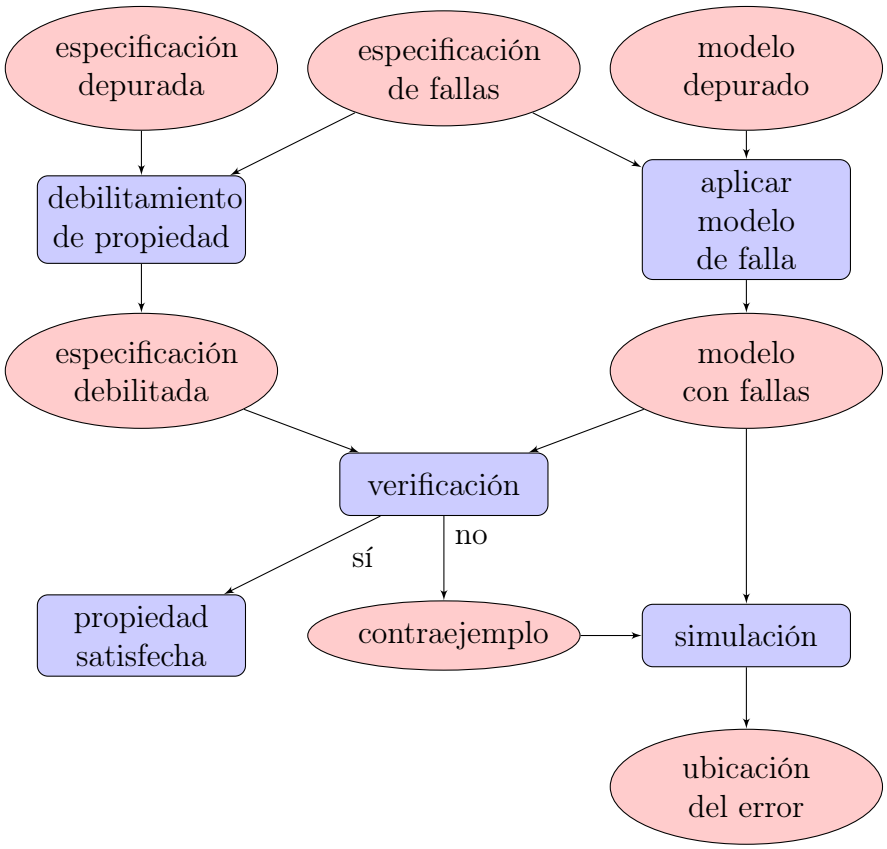

Figura 4.2: Proceso de model checking para sistemas con fallas

Calidad de la implementación. NuSMV está escrito en ANSI C, es compatible con POSIX y provee interfaces para el uso de distintos *SAT solvers*. El código es muy robusto, portable, eficiente y fácil de entender.

El código fuente de NuSMV se distribuye bajo la licencia LGPL 2.1 [17]. La elección de esta licencia facilita la adopción de la herramienta, y contribuye al desarrollo científico y a la introducción del model checking en entornos industriales [14].

4.2.1. El lenguaje de NuSMV

En esta sección, presentamos de manera sintética la sintaxis y la semántica del lenguaje de especificación de NuSMV que necesitamos para comprender

los ejemplos de capítulos siguientes. Para una referencia completa del lenguaje, recomendamos consultar el "Manual del Usuario de NuSMV 2.4" [12].

La verificación de un modelo en NuSMV dará *True* si el modelo satisface la propiedad especificada. De lo contrario, dará *False* junto a una traza que muestra una ejecución que hace falsa la propiedad.

El modelo de un sistema consta de uno o más módulos que se declaran con la palabra `MODULE` y el nombre del módulo. Es necesario que al menos uno de los módulos se llame `main`. Por ejemplo:

```
MODULE Sender
```

Cada módulo declara sus variables con sus tipos utilizando la palabra `VAR`. Por ejemplo:

```
VAR
  x : boolean;
  state : {Running, Ready, Blocked};
  i : 0..4;
```

En este ejemplo, la variable `x` tiene tipo booleano; `state` tiene el tipo enumerado con tres únicos valores: `Running`, `Ready` y `Blocked`; y la variable `i` está en el rango entero $[0, 4]$.

En NuSMV, el valor de verdad *False* se representa con 0 y *True*, con 1. Los operadores modales de LTL tienen los mismos nombres que los presentados en este trabajo, mientras que los operadores lógicos se muestran en la Tabla 4.1.

Operador Booleano	Operador NuSMV
$\neg$	`!`
$\wedge$	`&`
$\vee$	`|`
$\Rightarrow$	`->`
$\Leftrightarrow$	`<->`

Tabla 4.1: Operadores lógicos en NuSMV

Las asignaciones se declaran con la palabra `ASSIGN` y permiten dar valores iniciales a las variables (`init`) o definir el próximo valor (`next`) que tomará luego de realizar una transición. Los nuevos valores pueden definirse de manera no determinística si los encerramos entre llaves. Por ejemplo:

```
ASSIGN
  init(x) := 0;
```

```
next(x) := ! x;

init(state) := Ready;
next(state) :=
  case
    Running : {Ready, Blocked};
    Blocked : Ready;
    Ready : Running;
  esac;
```

En el ejemplo anterior, observamos que en el estado inicial `x` es 0. En cada transición, `x` va alternando su valor entre 0 y 1. También podemos observar una asignación por casos (`case`). Cada una de las guardas del `case` (a la izquierda de los ':') se evalúa ordenadamente desde arriba hacia abajo. Se asignará el valor a la derecha de los ':' de la primera guarda que sea verdadera. El `case` debe ser completo, es decir, la disyunción de todas las guardas debe ser verdadera.

Alternativamente, se pueden especificar los cambios de estado dando el valor inicial de las variables y las transiciones mediante fórmulas booleanas. Para ello, se utilizan las palabras `INIT` y `TRANS`, respectivamente. Por ejemplo:

```
INIT
  !effect & !occurrence
TRANS
  !effect & next(effect) <-> next(occurrence)
```

En su estado inicial, `effect` y `occurrence` tiene el valor de verdad falso. Habrá una transición de un estado actual p a un estado siguiente q, si el valor de `effect` en p y los valores de `effect` y `occurrence` en q satisfacen la fórmula dada en `TRANS`.

A veces, podemos querer utilizar un símbolo para referirnos al valor de una expresión (una macro). En este caso, el símbolo se define con la palabra `DEFINE`. Por ejemplo, la macro `m`:

```
DEFINE m := (x > y & y > z);
```

Las propiedades LTL se declaran con la palabra `LTLSPEC`, y las CTL, con `SPEC`. En el siguiente ejemplo, mostramos la propiedad de equidad fuerte:

```
LTLSPEC (G F p) -> (G F q)
```

Los módulos sirven como tipos abstractos de datos y pueden ser instanciados. Para ello, basta con declarar una variable que tenga como tipo el

nombre del módulo. Los módulos pueden tener parámetros, los cuales deben encerrarse entre paréntesis y separarse con comas. Se puede acceder a una variable de un módulo utilizando el operador '.' como se observa en la especificación del siguiente ejemplo:

```
MODULE counter
VAR
  i : 0..4;
ASSIGN
  init(i) := 0;
  next(i) :=
    case
      i < 4 : i + 1;
      i = 4 : 0;
    esac;

MODULE main
VAR
  c : counter;

LTLSPEC G c.i < 5
```

En NuSMV, los módulos se componen de manera sincrónica. Si queremos agregar no determinismo y hacer que se ejecuten de modo asincrónico, debemos anteponer la palabra `process` al nombre del módulo en la declaración de las instancia. Por ejemplo, podemos crear un proceso con el módulo `Sender` presentado antes:

```
sender : process Sender;
```

NuSMV implementa algunas de las restricciones de equidad mostradas en la sección 3.3. La equidad incondicional se implementa con el comando `JUSTICE`, y la equidad débil, con el comando `COMPASSION`. Por ejemplo:

```
JUSTICE p;
COMPASSION(p, q);
```

4.3. Observaciones finales

NuSMV no implementa equidad fuerte así que la agregamos en la biblioteca de fallas de nuestra herramienta. Su sintaxis es la siguiente:

```
STRONG_FAIRNESS(p, q);
```

Necesitamos la equidad fuerte para permitir a los procesos y a las fallas alternarse de modo tal que, tras la recuperación de una falla, el sistema pueda hacer alguna transición en los estados legales. Como dijimos en la sección 3.4, las restricciones de equidad no se pueden expresar en CTL; por lo tanto, sólo permitiremos el uso de propiedades LTL. FALLUTO no impone esta limitación y es responsabilidad del usuario comprobar que se cumpla.

Capítulo 5

Lenguaje de FALLUTO

> Language is not only the vehicle of thought, it
> is a great and efficient instrument in thinking.
>
> Humphry Davy

Como vimos en el capítulo 2, el modelo de fallas de un programa es una transformación de un programa que extiende sus transiciones originales para incorporar el comportamiento de una falla. También destacamos que el programa que resulta de dicha transición no es el que se implementa sino que es un instrumento para razonar sobre el comportamiento de las fallas y su interacción con el programa original.

En este capítulo, presentamos FALLUTO, una herramienta que extiende el lenguaje de NuSMV, y que permite *declarar* fallas y los efectos que éstas tienen sobre un programa[1]. El carácter declarativo del lenguaje de FALLUTO simplifica la actividad de modelado y permite al usuario concentrarse en el sistema por verificar sin preocuparse por los detalles del tratamiento de las fallas.

5.1. Los elementos de una falla

Antes de entrar en los detalles formales de la gramática de FALLUTO, comentaremos las actividades relacionadas con la identificación de fallas y su modelado. Luego, veremos cómo el resultado de estas actividades encaja en los elementos sintácticos incorporados por nuestra herramienta.

[1]Para ser más precisos, FALLUTO extiende un subconjunto propio del lenguaje de NuSMV, ya que deja afuera las especificaciones de tiempo real.

Generalmente, cuando pensamos en las fallas que pueden afectar un sistema, lo primero que hacemos es *identificarlas*. Por ejemplo, el sistema estará sujeto a un "pico de corriente", "daño en un enlace de red", "división por cero", "detención súbita", etc. Algunas fallas sólo pueden ocurrir en determinados momentos y bajo ciertas condiciones. Por ejemplo, una "división por cero" solamente podrá producirse cuando el sistema esté haciendo cálculos aritméticos, o un sistema sufrirá un "pico de corriente" cuando su fuente de alimentación esté operando. Por lo tanto, podemos decir que las fallas tienen una *precondición*. Las fallas también tienen un *efecto* sobre el sistema ya que pueden modificar su estado o inhabilitar alguna acción. Por ejemplo, un "pico de corriente" puede dejar una fuente de alimentación en un estado no operativo, y la "radiación cósmica" puede cambiar el valor de un registro en un satélite. Para *recuperarse* de esa situación, quizás sea necesaria la intervención humana (p. ej. sustituyendo un componente dañado) o tal vez el sistema pueda recuperarse en forma autónoma (p. ej. reintentando el envío de un paquete perdido). El lenguaje de FALLUTO nos obliga a pensar en todas estas cosas y agregarlas explícitamente en el modelo del sistema.

5.2. Declaración de fallas

En NuSMV, los módulos de un sistema están dados por sus variables, constantes y transiciones. En FALLUTO, también pueden incluir una o más fallas. La regla sintáctica para la declaración de una falla es:

```
fault :: FAULT identifier
           pre ( simple_expression ) ;
           effect ( effect_list ) ;
           restores ( next_expression ) ;
```

Descompongamos ahora esa regla gramatical y analicemos sus partes. Nos referiremos a `identifier` como el *nombre* de la falla. El nombre de una falla debe ser único por módulo y es cualquier secuencia alfanumérica válida como nombre de variable en NuSMV, salvo las nuevas palabras claves agregadas por FALLUTO:

```
FAULT, pre, effect, restores, disabled_by
```

Algunos ejemplos de nombres de fallas son `network_down`, `bit_flip` y `crash`. Los nombres de las fallas pueden coincidir con el de alguna variable. Sin embargo, no se puede declarar dos fallas con el mismo nombre: la herramienta detecta esta situación y notifica al usuario con un error.

La precondición para la ocurrencia de la falla (`pre`) debe expresarse como una `simple_expression` de NuSMV. Esto es una variable, una constante, una comparación aritmética o una operación lógica. Ejemplos de precondiciones posibles son `(x > 5 & y < 3)` y (`!z`). Nótese que el predicado *true* es una precondición posible, lo escribimos con la constante `1` y corresponde a la falla que puede ocurrir en cualquier momento.

El efecto de la ocurrencia de una falla (`effect`) es una lista con el nuevo estado del sistema. Su regla sintáctica es la siguiente:

```
effect_list :: var_id = simple_expression
             |  var_id = simple_expression , effect_list
```

No es necesario dar el nuevo valor de todas las variables, sino solamente el de aquellas que cambian tras la ocurrencia de la falla. Algunos ejemplos son `(up = 0)`, `(x = 1, y = 0)`. Cabe señalar que también es posible indicar el no determinismo del nuevo valor de una variable. Por ejemplo, si la variable `x` puede tomar el valor *true* o *false*, entonces escribimos `(x = {0, 1})`. También podemos usar rangos para referirnos al nuevo valor de una variable. Por ejemplo, `(x = 0..255)` indica que `x` puede tomar cualquier valor entre 0 y 255 indistintamente.

La restauración del sistema a un estado sin falla (`restores`) se declara con una `next_expression` de NuSMV y es una `simple_expression` en la que se puede usar el operador `next`. Por ejemplo, si el sistema se restaura si el valor de `x` en el próximo estado es mayor que 3, entonces escribiremos `next(x) > 3`. Si tras la ocurrencia de la falla, el sistema no puede recuperarse, entonces el predicado corresponderá escribir el predicado *false* con la constante `0`.

Veamos un ejemplo completo de la declaración de una falla:

```
FAULT crash
  pre(up);
  effect(up = 0);
  restores(0);
```

Este caso corresponde a una falla a la que llamamos `crash`. Esta falla sólo ocurrirá cuando la variable `up` sea *true* y cambiará su valor a *false*. El sistema no se recuperará tras la ocurrencia de esta falla.

Las reglas gramaticales presentadas hasta aquí nos permiten declarar una falla, la condición bajo la cual puede ocurrir, el efecto que tiene sobre el sistema y en qué condición éste se puede recuperar. En la próxima sección, veremos cómo indicar que una falla inhabilita una transición en el sistema.

5.3. Inhabilitación de transiciones

En FALLUTO, las transiciones del sistema están dadas por las asignaciones que usan el operador `next`. En FALLUTO, tenemos una nueva regla sintáctica para las asignaciones:

```
next_assign :: next ( var_id ) := next_expression ;
             | next ( var_id ) := next_expression
                 disabled_by { fault_names_list } ;
```

donde `fault_names_list` corresponde a una lista de nombres de fallas separadas por comas:

```
fault_names_list :: identifier
                  | identifier , fault_names_list
```

El comando `disabled_by` indica que la transición queda inhabilitada mientras persista el efecto de alguna de las fallas en la lista, i.e. desde la ocurrencia de la falla y hasta que el sistema se recupere. Los nombres de las fallas que aparecen en una lista deben ser conocidos por FALLUTO, es decir, las fallas deben haber sido declaradas con anterioridad. En caso contrario, se notifica al usuario con un mensaje de error.

Veamos un par de ejemplos de transiciones sencillas que quedan inhabilitadas por la ocurrencia de fallas:

```
next(x) := ! x disabled_by {f};

next(i) :=
  case
    i < 4 : i + 1;
    i = 4 : 0;
  esac disabled_by {f1, f2, f3};
```

Bajo condiciones normales, la variable `x` va alternando su valor entre *true* y *false* en cada transición, y la variable `i` implementa un contador módulo cinco. Sin embargo, las transiciones de `x` permanecen inhabilitadas desde que ocurre la falla de nombre `f` y hasta que el sistema se recupera, mientras que las de `i` permanecen inhabilitadas bajo el efecto de las fallas `f1`, `f2` o `f3`.

Finalmente, el último cambio introducido en la sintaxis de NuSMV ofrece un mayor control sobre las transiciones ya que permite señalar como inhabilitadas sólo algunas acciones individuales en el cuerpo de una expresión con casos (`case_expr`):

```
case_expr :: case case_body esac
```

```
case_body :: case_element ;
           | case_body case_element ;

case_element :: simple_expr : simple_expr
              | simple_expr : simple_expr
                  disabled_by { fault_names_list }
```

Nótese que el ejemplo anterior del contador módulo cinco también puede escribirse de la siguiente manera:

```
next(i) :=
  case
    i < 4 : i + 1 disabled_by {f1, f2, f3};
    i = 4 : 0 disabled_by {f1, f2, f3};
  esac;
```

Sin embargo, como se puede deducir de las reglas sintácticas, no es necesario que ambos casos se vean inhabilitados por la ocurrencia de esas fallas. Es más, algunas fallas pueden afectar un caso y no otro. Por ejemplo:

```
next(i) :=
  case
    i < 4 : i + 1 disabled_by {f1};
    i = 4 : 0 disabled_by {f2, f3};
  esac;
```

5.4. Observaciones finales

La motivación para incorporar un lenguaje declarativo para el modelado de fallas surge de la necesidad de simplificar la tarea del usuario final. Al evitar cualquier indicación explícita de la ocurrencia de las fallas, no es necesario hacer grandes modificaciones al modelo original para verificar su tolerancia a fallas.

La presentación de la gramática de FALLUTO en este capítulo hace hincapié en la comprensión intuitiva de los elementos sintácticos introducidos y deja de lado los detalles de la sintaxis de NuSMV. La gramática formal completa se puede encontrar en el Apéndice B.

En el próximo capítulo, se describe la arquitectura de FALLUTO y cómo convierte las construcciones del nuevo lenguaje en un programa que NuSMV pueda entender.

Capítulo 6

Funcionamiento de FALLUTO

> I think in all fairness I should explain to you exactly what it is that I do.
>
> Nicky Santoro, *Casino*

El capítulo anterior nos mostró las nuevas construcciones que FALLUTO agrega al lenguaje de NuSMV para modelar las fallas. En este capítulo, veremos cómo se compilan esas producciones sintácticas en un modelo y una especificación verificables por NuSMV. Además, presentaremos los otros componentes de FALLUTO que son distribuidos junto con el compilador: el intérprete de traza, que procesa las especificaciones y los contraejemplos para que sólo incluyan objetos declarados por el usuario, y el visualizador de traza, que genera un autómata que facilita la comprensión de los contraejemplos. Para poder comprender el funcionamiento completo de la herramienta, primero daremos una visión de la arquitectura de FALLUTO y luego pasaremos a los detalles de la compilación.

6.1. Arquitectura de la herramienta

La arquitectura de FALLUTO sigue el patrón de tubos y filtros [39] (ver Figura 6.1). La herramienta está formada por los siguientes componentes (filtros):

Compilador. Dado un modelo y una especificación escritos en el lenguaje de FALLUTO (Capítulo 5), lo convierte a código de NuSMV.

Model Checker. NuSMV se usa como motor de verificación y toma la salida del compilador para generar una traza.

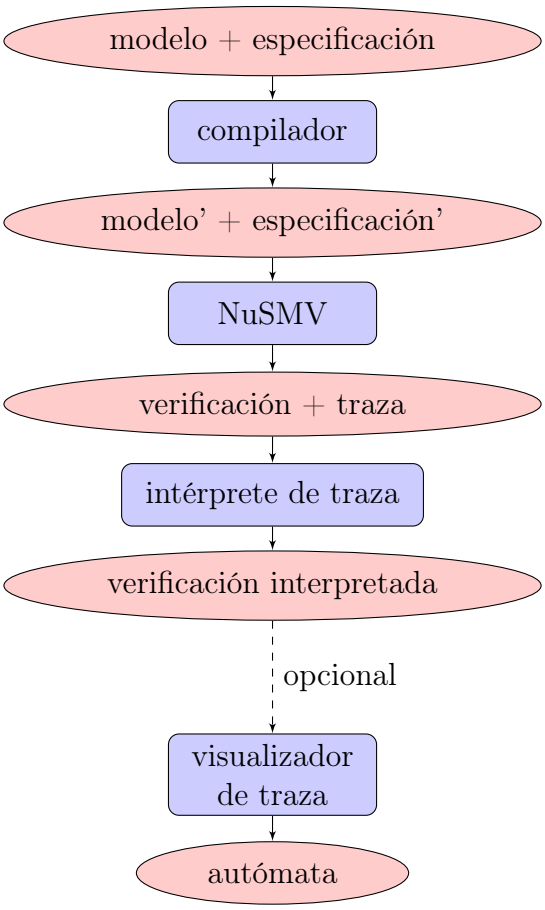

Figura 6.1: Arquitectura de FALLUTO

Intérprete de traza. Interpreta la traza generada por NuSMV utilizando objetos del modelo original y quitando cualquier referencia a los objetos de soporte que crea el compilador para las fallas.

Visualizador de traza. Permite generar un autómata a partir de una traza de contraejemplo, facilitar la comprensión de las trazas y ayudar al usuario en una simulación.

Las salidas de cada etapa se guardan en archivos que pueden ser recuperados y analizados con posterioridad por el usuario. En particular, cada componente toma como entrada la salida del componente anterior y la procesa de un modo completamente lineal tal como en el patrón clásico.

6.2. Compilación a NuSMV

En esta sección, damos una abstracción sobre los módulos de FALLUTO y NuSMV y luego describimos detalladamente cómo es la función de compilación. Luego usaremos estas ideas para describir como se compila nuestro lenguaje. La implementación concreta de FALLUTO agrega algunas macros para simplificar la generación automática de expresiones. Sin embargo, el algoritmo es idéntico al presentado en este informe.

6.2.1. Módulos

Un módulo de NuSMV es $M_{NuSMV} = \langle N, A, V, D, I, T \rangle$, donde N es el nombre del módulo, A, la lista de argumentos, V, el conjunto de variables, D, el conjunto de definiciones (macros), e I, el conjunto de estados iniciales.

En FALLUTO, extendemos la idea de módulo para contemplar las fallas. Entonces, tenemos que $M_{\text{FALLUTO}} = \langle N, A, V, D, I, T, F \rangle$, donde N es el nombre del módulo, A, la lista de argumentos, V, el conjunto de variables, D, el conjunto de definiciones (macros), I, el conjunto de estados iniciales, y F, el conjunto de fallas.

6.2.2. La función *compilar*

Definimos *compilar* como la transformación de módulos de FALLUTO a listas de módulos de NuSMV, de la siguiente manera:

$$\begin{aligned} &\mathit{compilar} \;::\; M_{\text{FALLUTO}} \rightarrow [M_{NuSMV}] \\ &\mathit{compilar}\ \langle N, A, V, D, I, T, F \rangle \;= \langle N, A, V', D', I, T' \rangle : [\ M_{N,f} \mid f \in F\ ] \end{aligned}$$

Veamos ahora qué son cada uno de los elementos de la definición anterior. En primer lugar, podemos notar que el nombre del módulo se preserva en la compilación. Dado que el nombre (N) elegido por el usuario es representativo en su sistema original, desearemos preservarlo.

Por cada falla f a la que esté sujeto el módulo, agregamos una variable v_f de tipo *process* $M_{N,f}$ y vfair_f de tipo booleano. La variable v_f es una instancia de la falla que se ejecuta de manera asíncrona con el módulo. La variable de equidad vfair_f nos permitirá forzar el requerimiento de equidad fuerte entre el módulo y su falla. Por lo tanto, nos queda que el nuevo conjunto de variables es:

$$V' \;=\; V \;\cup\; \{\ v_f \mid f \;\in\; F\ \} \;\cup\; \{\ \mathit{vfair}_f \mid f \;\in\; F\ \}$$

Por cada falla f, agregamos dos nuevas macros al conjunto D: pre_f y $restore_f$, que son iguales a la precondición de f y su condición de restauración respectivamente. Nos queda entonces:

$$D' = D \cup \{ pre_f \mid f \in F \} \cup \{ restore_f \mid f \in F \}$$

El conjunto de transiciones del módulo compilado tiene las transiciones del módulo original que no están sujetas a fallas, una transformación de cada transición inhabilitada por fallas y la transición correspondiente a cada una de las nuevas variables $vfair_f$. Denotamos con t_v la transición de la variable v y tenemos entonces el siguiente conjunto de transiciones:

$$\begin{aligned} T' = & \{ t \mid t \in T,\ t \text{ no está inhabilitada por fallas} \} \\ & \cup \{ t' \mid t \in T,\ t \text{ está inhabilitada por fallas} \} \\ & \cup \{ t_{vfair_f} \mid \text{para cada } vfair_f \in V' \} \end{aligned}$$

Para explicar cómo se transforman las transiciones con fallas (t'), consideraremos dos situaciones: el comando *disabled_by* está fuera de una asignación por casos o dentro de ésta. En la primera situación, tenemos algo como

$$next(v) := expr\ disabled\_by\ \{ f_1, \ldots, f_n \};$$

Recordemos del capítulo anterior, que el sistema sólo podrá hacer una transición hacia un estado donde v es igual a *expr*, cuando no esté bajo el efecto de alguna de las fallas $f_1, ..., f_n$. En caso contrario, en el siguiente estado, la variable no cambia su valor. Usaremos la variable booleana $v_f.active$ para indicar que el sistema está bajo el efecto de v_f (la falla f ocurrió y el sistema todavía no se recuperó – ver subsección 6.2.3). Entonces tenemos que la transición anterior se transforma en:

$$\begin{aligned} next(v) := & \ case \\ & \neg v_{f_1}.active \wedge \cdots \wedge \neg v_{f_n}.active \quad : expr; \\ & 1 \quad : v; \\ & esac; \end{aligned}$$

Veamos cómo se transforma una transición cuando la inhabilitación ocurre

dentro de una asignación por casos. La transición original será de la forma:

$$
\begin{array}{l}
next(v) := case \\
\qquad\qquad c_1 \ : \ expr_1; \\
\qquad\qquad \ldots \\
\qquad\qquad c_i \ : \ expr_i \ disabled\_by \ \{ \ f_1, \ldots, f_n \ \}; \\
\qquad\qquad \ldots \\
\qquad\qquad c_n \ : \ expr_n; \\
\qquad\quad esac;
\end{array}
$$

Siguiendo un razonamiento similar al anterior, la acción correspondiente a c_i sólo estará habilitada si ninguna de las fallas ocurre. Por lo tanto, la asignación se convierte en:

$$
\begin{array}{ll}
next(v) := case & \\
\qquad\qquad c_1 & : expr_1; \\
\qquad\qquad \ldots & \\
\qquad\qquad c_i \wedge \neg v_{f_1}.active \wedge \cdots \wedge \neg v_{f_n}.active & : expr_i; \\
\qquad\qquad \ldots & \\
\qquad\qquad c_n & : expr_n; \\
\qquad\qquad 1 & : v; \\
\qquad\quad esac; &
\end{array}
$$

Nótese que agregamos el caso $1 : v$; ya que NuSMV exige que la asignación por casos sea una función total.

Para terminar de definir T' completamente, debemos ver cómo son las transiciones de las variables de equidad (t_{vfair_f}). Esta variable sirve para asegurar que, cuando el sistema opera normalmente (sin fallas), entonces tiene la posibilidad de ejecutar alguna de las transiciones potencialmente afectadas por fallas. La variable t_{vfair_f} tomará el valor *true* cuando la falla f no esté activa y se satisfaga la condición de alguna guarda afectada por f. Si la falla f ocurre, la variable toma el valor $false$:

$$
\begin{array}{ll}
next(vfair_f) := case & \\
\qquad\qquad (c_1 \vee \cdots \vee c_n) \wedge \neg v_f.active \wedge \neg vfair_f & : 1; \\
\qquad\qquad 1 & : 0; \\
\qquad\quad esac; &
\end{array}
$$

donde c_1, ..., c_n son las guardas de las transiciones inhabilitadas por la falla f. Nótese que si ninguna asignación por casos está afectada por fallas,

entonces la disjunción $(c_1 \vee \cdots \vee c_n)$ se elimina por rango vacío. En la subsección 6.2.4 mostramos cómo usaremos esta variable para considerar sólo ejecuciones fuertemente equitativas.

6.2.3. Módulos de fallas

El módulo $M_{N,f}$ se crea para reflejar el comportamiento de cada falla f en el módulo de nombre N. Se instancia como un proceso asíncrono, maneja la ocurrencia de f y modifica los valores de las variables que participan del efecto de la falla. Veamos cuáles son los elementos de $M_{N,f} = \langle N_f, A_f, V_f, \varnothing, I_f, T_f \rangle$.

La lista de argumentos A_f tiene una referencia a la precondición de f y a su condición de restauración para poder determinar cuándo puede ocurrir la falla y cuándo se recupera el sistema. Además, tiene una referencia a cada una de las variables del módulo de nombre N que son afectadas por f. De este modo, la falla puede alterar sus valores en alguna de sus transiciones. Por lo tanto, nos queda que A_f es

$$A_f \;=\; pre_f : restore_f : [\; v \mid v \text{ participa en el efecto de } f \;]$$

El módulo de falla tiene una sola variable: *active* de tipo booleano. Esta variable toma el valor 1 (*true*) cuando la falla ocurre y mientras el sistema está bajo falla. Cuando el sistema se recupera, *active* se hace 0 (*false*). En el estado inicial, el sistema no está bajo el efecto de fallas y entonces tenemos

$$V_f \;=\; \{\; active \;\}$$
$$I_f \;=\; \{\; \neg active \;\}$$

Veamos ahora cómo son las transiciones en el módulo de falla. Como dijimos anteriormente, *active* será 1 cuando la falla ocurra. La falla sólo podrá ocurrir en aquellos estados donde se cumpla la precondición, pero su ocurrencia será no determinística: el sistema no fallará cada vez que se dé la precondición. Además, asumiremos que mientras el sistema está bajo una falla, ésta no vuelve a ocurrir. Por otro lado, la variable *active* permanecerá con valor 1 hasta que se cumpla la condición de restauración. En ese momento, la falla desaparece y el sistema vuelve a sus transiciones normales. Luego, la transición de *active* (llamémosla t_{active}) es

$$\begin{array}{l} next(active) := case \\ \qquad\qquad pre \wedge \neg active \qquad\qquad : \{\; 0, 1 \;\}; \\ \qquad\qquad active \wedge next(restore) \qquad : 0; \\ \qquad\qquad 1 \qquad\qquad : active; \\ \qquad esac; \end{array}$$

Para cada variable v que participa del efecto de la falla f, supongamos que e_v es la expresión del efecto de f sobre v. Entonces, si en el estado siguiente se produce la falla, el valor de v será la expresión e_v, y la transición que agregamos es la siguiente:

$$
\begin{array}{l}
next(v) := case \\
\qquad\qquad \neg active \wedge next(active) \qquad\qquad : e_v; \\
\qquad\qquad 1 \qquad\qquad : v; \\
\qquad esac;
\end{array}
$$

Finalmente, tenemos entonces que el conjunto de transiciones del módulo de falla es

$$T_f \;=\; \{\; t_{active} \;\} \;\cup\; \{\; t_v \mid \text{para cada variable } v \text{ afectada por } f \;\}$$

6.2.4. Compilación de las especificaciones

Vimos que FALLUTO modela cada falla como un proceso que se ejecuta en paralelo con los definidos por el usuario. Como podemos ver en la Figura 6.2, es posible encontrar una planificación donde las fallas ocurren constantemente y los procesos nunca se ejecutan. Esa planificación sirve al model checker como contraejemplo de cualquier propiedad interesante: el sistema no progresa y ninguna propiedad puede satisfacerse.

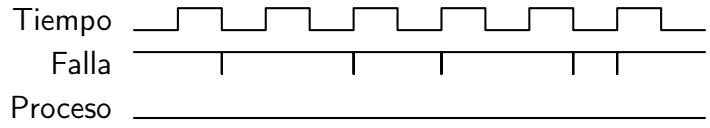

Figura 6.2: Planificación no equitativa de procesos y fallas

Es necesario entonces considerar sólo aquellas planificaciones con equidad fuerte entre procesos y fallas, ya que son las únicas en las cuales puede haber progreso (ver Figura 6.3).

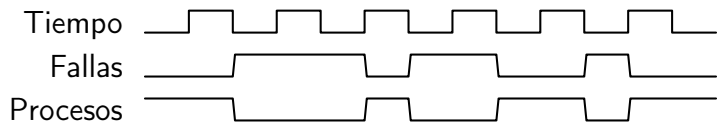

Figura 6.3: Planificación equitativa de procesos y fallas

Para poder asegurar equidad fuerte, pediremos que si un proceso puede ejecutar alguna transición infinitamente a menudo, entonces la variable de equidad debe tomar el valor *true* infinitamente a menudo. Así nos aseguramos que los procesos tienen posibilidad de progresar y las fallas no los interrumpen constantemente (Figura 6.2).

Entonces, debilitaremos la propiedad por verificar del siguiente modo. Sean,

- F, el conjunto de las fallas a la que está sujeto un módulo,
- para cada $f \in F$, $active_f$, la variable que indica la ocurrencia de f, y
- ψ, la propiedad que se desea verificar,

pedimos que la propiedad se satisfaga en todas las ejecuciones con equidad fuerte entre los procesos y las fallas. Formalmente, la especificación LTL está dada por la siguiente fórmula:

$$\left(\bigwedge_{f \in F} STRONG\_FAIRNESS((c_1 \vee \cdots \vee c_n) \wedge \neg active_f,\ vfair_f) \right) \implies \psi$$

6.3. Interpretación de la traza

Como mostramos en la sección anterior, FALLUTO genera un nuevo modelo con el agregado de módulos y variables auxiliares que permiten representar las fallas y sus ocurrencias. Además, la herramienta agrega el requerimiento de equidad fuerte a la especificación. Por lo tanto, el model checker tomará una representación del sistema sustancialmente distinta a la ingresada por el usuario y todas sus salidas contendrán objetos que no aparecían en el modelo original. Para resolver este problema, FALLUTO incorpora un intérprete de traza que elimina casi todos los objetos que no corresponden al modelo de entrada y corrige la especificación para que sea la misma que intentó verificar el usuario (sólo se deja la variable que señaliza la ocurrencia de la falla para ayudar a encontrar el error en el modelo). También se incluye un visualizador de traza que convierte la representación textual del contraejemplo en un autómata. Este autómata facilita el seguimiento de los estados que hacen falsa la especificación y es muy útil para las simulaciones de las trazas en NuSMV.

6.4. Implementación de la herramienta

Todos los componentes de FALLUTO (el compilador, el intérprete y el visualizador de traza) están implementados usando Ruby [29], un lenguaje

de programación con soporte para múltiples paradigmas entre ellos funcional, orientado a objetos e imperativo.

El compilador sale del esquema convencional de lexers y parsers gracias al uso de *Parsing Expression Grammars* (PEG) [16]. Las PEG son un formalismo más potente que las gramáticas libres de contexto que pueden convertirse en un parser de tipo *packrat*. El uso de PEG con Ruby está resuelto mediante el paquete Treetop [36] que permite definir las reglas gramaticales con una sintaxis muy sencilla. En el Apéndice A, se puede encontrar una explicación más completa sobre este formalismo y su implementación en Treetop.

El árbol sintáctico es recorrido mediante el patrón Iterator [18], y el compilador implementa el patrón Visitor [18] con un método para convertir cada nodo del árbol.

6.4.1. Limitaciones de Falluto

La versión actual de Falluto tiene algunas limitaciones:

- Las especificaciones están restringidas al uso de LTL por los problemas de fairness.
- Ya que hacemos restricciones sobre la equidad de los procesos y las fallas, los módulos deben estar instanciados como procesos para poder ser verificados.
- El uso de TRANS no es compatible con el lenguaje de fallas. La combinación de ASSIGN y TRANS en un mismo programa tiene resultados indefinidos.
- Es posible tener colisiones de nombres entre las variables generadas automáticamente y las variables o macros definidas por el usuario.
- Falluto no detecta algunos errores estáticos que aparecen en tiempo de compilación (errores de tipos, variables no definidas, etc). Dichos errores son manejados por NuSMV, y por lo tanto, es posible que los números de línea de error en el programa compilado no coincidan con los del programa original.
- Es responsabilidad del usuario controlar el impacto de estas limitaciones en el uso de la herramienta.

Capítulo 7

Modelos de fallas

Como explicamos en el capítulo 2, un modelo de fallas es una transformación de un programa en otro que describe cómo afectan las fallas a las transiciones. En este capítulo, presentamos distintos modelos de fallas y sus descripciones usando FALLUTO.

En los ejemplos a continuación, asumimos que $\mathcal{F}$ es el conjunto de procesos sujetos al modelo de falla y que C es un canal de comunicación entre dos procesos.

7.1. *Crash*

El modelo de falla *crash* describe la detención abrupta de un proceso [23, 24]. Si pensamos en el programa p como un autómata, este modelo agrega un nuevo estado final *crashed* y una transición desde cada estado hacia el estado *crashed*. Formalmente, este modelo se describe de la siguiente manera:

Para cada proceso $f \in \mathcal{F}$:

C1 Se agrega una variable booleana $stopped_f$ que denotará que el proceso se ha detenido.

C2 Se transforma cada acción a en

$$\langle \text{crash } a \rangle :: \neg stopped_f \wedge \langle a \rangle$$

para inhabilitar las acciones en caso de que el proceso se detenga.

C3 Se agrega la acción

$$\langle \text{crash} \rangle :: true \rightarrow stopped_f := true$$

para indicar que en cualquier momento puede ocurrir una avería que detenga al proceso.

7.1.1. Representación en FALLUTO

La falla de tipo *crash* tiene las siguientes características (a) puede ocurrir en cualquier momento, (b) su ocurrencia no afecta el valor de las variables del sistema, y (c) no puede ser recuperada.

```
FAULT crash
  pre(1);
  effect();
  restores(0);
```

Usaremos el siguiente modelo de un contador módulo 5 para transformarlo con la falla de tipo crash.

```
MODULE counter
VAR
  i : 0..4;
ASSIGN
  init(i) := 0;
  next(i) :=
    case
      i < 4 : i + 1;
      i = 4 : 0;
    esac;
```

La falla de tipo crash inhabilita todas las transiciones del contador, por lo tanto debemos agregar el comando `disabled_by` en cada una de ellas:

```
MODULE counter
FAULT crash
  pre(1);
  effect();
  restores(0);
VAR
  i : 0..4;
ASSIGN
  init(i) := 0;
  next(i) :=
    case
      i < 4 : i + 1 disabled_by {crash};
      i = 4 : 0 disabled_by {crash};
    esac;
```

7.2. *Fail-stop*

Un procesador es un circuito electrónico que implementa un conjunto de instrucciones que permiten realizar operaciones aritméticas, lógicas y de entrada/salida. Cada una de estas instrucciones altera el estado interno del procesador o de sus dispositivos periféricos según una semántica precisamente definida por el fabricante del circuito.

Un procesador de tipo *fail-stop* [35] es un modelo teórico con un comportamiento muy sencillo. Todo su estado interno y parte de almacenamiento son volátiles y sus contenidos se pierden en caso de avería. El almacenamiento restante se considera estable, y su contenido no es afectado por ningún tipo de avería. A diferencia de un procesador real, el procesador fail-stop nunca cambia erróneamente su estado interno. En caso de avería, simplemente detiene la ejecución del programa y pierde su estado interno y el contenido de la memoria volátil.

En el modelo de falla *fail-stop*, un proceso se detiene y este hecho es detectable por otros procesos. Además, cualquier información escrita por el proceso fallido está disponible en el almacenamiento estable del procesador y puede ser accedida por los demás procesos.

7.2.1. Representación en Falluto

El siguiente ejemplo modela un procesador *fail-stop* muy sencillo que alterna el valor de *b*. Por su parte, la variable *state* indica si el procesador está en modo operativo o detenido. Inicialmente, su valor es *Operational*, pero al detectarse un error, cambia a *Stopped*.

```
MODULE FailStopProcessor
VAR
  b : boolean;
  state : {Operational, Stopped};
ASSIGN
  init(b) := 0;
  next(b) :=
    case
      state = Operational : !b;
      1                   : b;
    esac;
  init(state) := Operational;
  next(state) := state;
```

La falla de tipo *fail-stop* puede ocurrir en cualquier momento. Como en el caso de la falla de tipo crash, el sistema no se puede recuperar de esta falla.

```
FAULT FailStopFault
  pre(state = Operational);
  effect(state = Stopped);
  restores(0);
```

Ahora transformamos el modelo del procesador para incluir la falla.

```
MODULE FailStopProcessor
FAULT FailStopFault
  pre(state = Operational);
  effect(state = Stopped);
  restores(0);
VAR
  b : boolean;
  state : {Operational, Stopped};
ASSIGN
  init(b) := 0;
  next(b) :=
    case
      state = Operational : !b;
      1                   : b;
    esac;
  init(state) := Operational;
  next(state) := state;
```

7.3. Omisión de envío

Este modelo de fallas describe la falla en que el envío de un mensaje m se completa, pero m no se inserta en el buffer de salida [23]. Su definición formal es la siguiente [20]:

Para cada proceso $f \in \mathcal{F}$ y cada canal C:

S1 Por cada acción de la forma

$$\langle x \rangle :: \langle \text{guarda}_x \rangle \rightarrow x_1, \ldots, x_n, C := y_1, \ldots, y_n, \mathbf{snd}(C, m)$$

se agrega la acción que modela la omisión del envío (el proceso puede elegir la acción en que no se envía el mensaje m):

$$\langle \text{s-omit } x \rangle :: \langle \text{guarda}_x \rangle \rightarrow x_1, \ldots, x_n := y_1, \ldots, y_n$$

7.3.1. Representación en FALLUTO

Modelamos un canal como un par (arr, msg) donde arr representa el arribo de un mensaje y msg, el mensaje. Inicialmente, no hay mensajes en el canal, así que el modelo en FALLUTO es el siguiente:

```
MODULE Canal
VAR
  arr : boolean;
  msg : 0..255;
INIT
  ! arr
```

En el transmisor, modelamos el envío del mensaje m asignando el valor m en la variable msg y el arribo de un mensaje poniendo arr en verdadero (i.e. en 1).

```
MODULE Tranmisor(c)
-- ...
ASSIGN
-- ...
  next(c.arr) :=
    case
      guarda_x : 1;
      1        : c.arr;
    esac;

  next(c.msg) :=
    case
      guarda_x : m;
      1        : c.msg;
    esac;
```

La omisión de envío es un falla que tiene las siguientes características: (a) puede suceder en cualquier momento, (b) no afecta el valor de las variables del sistema, y (c) el sistema se recupera "solo":

```
FAULT s_omit
  pre(1);
  effect();
  restores(1);
```

Ahora transformamos el transmisor con el modelo de la omisión de envío. En este caso, tanto la notificación del arribo del mensaje como la escritura

del mensaje en el canal son inhabilitadas:

```
MODULE Tranmisor(c)
FAULT s_omit
  pre(1);
  effect();
  restores(1);
-- ...
ASSIGN
-- ...
  next(c.arr) :=
    case
      guarda_x : 1 disabled_by {s_omit};
      1        : c.arr;
    esac;

  next(c.msg) :=
    case
      guarda_x : m disabled_by {s_omit};
      1        : c.msg;
    esac;
```

Notar que es opcional la inhabilitación de la asignación del valor siguiente de *c.msg* ya que un proceso notará que hay un nuevo mensaje en el canal cuando *c.arr* sea verdadero.

7.4. Omisión de recepción

Este modelo de fallas describe la falla en que el mensaje m se inserta en el buffer de entrada de un proceso y éste no lo recibe [23]. Su definición formal es la siguiente [20]:

Para cada proceso $f \in \mathcal{F}$ y cada canal C:

R1 Por cada acción de la forma

$$\langle x \rangle :: \langle \text{guarda}_x \rangle \wedge \mathbf{arr}(C, m) \rightarrow \\ x_1, \ldots, x_n, z, C := y_1, \ldots, y_n, \mathbf{rcv}(C, m), \mathbf{rmv}(C, m)$$

se agrega la acción que modela la omisión de la recepción (se quita el mensaje m del buffer y el proceso no lo recibe):

$$\langle \text{r-omit } x \rangle :: \langle \text{guarda}_x \rangle \wedge \mathbf{arr}(C, m) \rightarrow \\ x_1, \ldots, x_n, C := y_1, \ldots, y_n, \mathbf{rmv}(C, m)$$

7.4.1. Representación en Falluto

El canal se modela igual que en el caso de la omisión de envío. El receptor asigna a la variable z el mensaje contenido en el canal.

```
MODULE Receptor(c)
-- ...
ASSIGN
-- ...
  next(c.arr) :=
    case
      guarda_x & c.arr : 0;
      1                : c.arr;
    esac;

  next(z) :=
    case
      guarda_x & c.arr : c.msg;
      1                : c.arr;
    esac;
```

La falla se modela de forma similar al caso anterior.

```
FAULT r_omit
  pre(1);
  effect();
  restores(1);
```

Finalmente, veamos cómo es la transformación del receptor cuando se inhabilita la lectura del canal.

```
MODULE Receptor(c)
FAULT r_omit
  pre(1);
  effect();
  restores(1);
-- ...
ASSIGN
-- ...
  next(c.arr) :=
    case
      guarda_x & c.arr : 0;
      1                : c.arr;
    esac;
```

```
next(z) :=
  case
    guarda_x & c.arr : c.msg disabled_by {r_omit};
    1                : c.arr;
  esac;
```

7.5. Omisión general

Este modelo de falla corresponde a un proceso que puede experimentar fallas al enviar y al recibir mensajes [32]. Puede verse como una combinación de la omisión de envío y recepción. Por lo tanto, su definición formal está dada por:

Para cada proceso $f \in \mathcal{F}$ y cada canal C:

G1 Aplicar la regla S1.

G2 Aplicar la regla R1.

La representación en FALLUTO puede deducirse a partir de los modelos anteriores.

7.6. Falla bizantina

Imaginemos que varias divisiones del ejército bizantino acampan en las afueras de una ciudad enemiga y que cada una de ellas es comandada por un general. Los generales pueden comunicarse entre ellos utilizando sólo un mensajero. Luego de observar el comportamiento del enemigo, los generales deben ponerse de acuerdo en un plan de acción común para atacar todos en el mismo momento. Sin embargo, algunos de los generales pueden ser traidores e intentar evitar que los generales leales lleguen a un consenso. Por ejemplo, los generales traidores podrían matar a un mensajero leal o enviar un mensajero con un mensaje incorrecto. Sin embargo, los generales necesitan un mecanismo que les garantice que todos lleguen al mismo acuerdo y que un pequeño número de traidores no los haga elegir un mal plan [25].

En un programa, algunos procesos pueden ser poco fiables, tal como algunos de los generales bizantinos, y fallar de modo inesperado (p. ej. no hacer una transición o hacer transiciones indebidas). Este modelo de falla se conoce como falla bizantina y está formalmente definido de la siguiente manera.

Para cada proceso $f \in \mathcal{F}$ y cada canal C:

B1 Se agrega una acción que no altera el estado del programa

$$\langle \text{loop} \rangle :: true \rightarrow \textbf{skip}$$

B2 Para cada mensaje $m \in M$, se agregan las acciones

$$\langle \text{send } m \rangle :: true \rightarrow C := \textbf{snd}(C, m)$$
$$\langle \text{recv } m \rangle :: \textbf{arr}(C, m) \rightarrow C := \textbf{rmv}(C, m)$$

7.6.1. Representación en FALLUTO

Primero, trataremos de enteder cómo se representa cada una de las acciones de este modelo de falla y luego presentaremos el modelo completo.

La acción *loop* es aquella que preserva el estado del programa (i.e. en el próximo estado, el valor de cada variable es igual al valor en el estado actual). Supongamos que $V = \{v_0, ..., v_{n_v}\}$ es el conjunto de variables del programa, tenemos que:

$$loop \equiv \bigwedge_{v \in V} next(v) = v$$

En FALLUTO, podemos escribir la lista de efectos correspondiente a *loop* como:

```
v_0 = v_0, ..., v_nv = v_nv
```

Donde usamos '...' para representar al resto de las variables.

La acción *send* envía algún mensaje por algún canal. Supongamos que $C = \{c_0, ..., c_{n_c}\}$ es el conjunto de los canales del programa y $M = \{m_0, ..., m_{n_m}\}$, el de mensajes, tenemos que:

$$send \equiv \bigvee_{v \in V, c \in C} next(c.msg) = m \wedge next(c.arr)$$

La lista de efectos correspondientes a fallas bizantinas en el envío es:

```
c_0.msg = { m_0, ..., m_nm }, c_0.arr = 1,
...,
c_nc.msg = { m_0, ..., m_nm }, c_nc.ar = 1
```

Por último, la acción *recv* quita algún mensaje del buffer de entrada de algún canal:

$$recv \equiv \bigvee_{c \in C} \neg next(c.arr)$$

Esto equivale a tener una lista de efectos donde la variable *arr* es 0:

```
c_0.arr = 0, ..., c_nc.arr = 0
```

Armados de todas estas listas de efectos, ya podemos definir el modelo de la falla bizantina. Esta falla puede ocurrir en cualquier momento y el sistema se recupera solo:

```
FAULT bizantina
  pre(1);
  effect( v_0 = v_0, ..., v_nv = v_nv,
          c_0.msg = { m_0, ..., m_nm },
          ...,
          c_nc.msg = { m_0, ..., m_nm},
          c_0.arr = {0, 1}, ..., c_nc = {0, 1});
  restores(1);
```

Nótese que usamos no determinismo para el valor de *arr* en cada canal. Esto surge de la combinación del efecto en el envío y la recepción.

7.7. Falla de enlace

Un canal de comunicaciones puede fallar y no llevar el mensaje desde el buffer de salida de un proceso al buffer de entrada de otro [23]. Por ejemplo, el mensaje podría perderse por un daño físico en el enlace. Tomemos como punto de partida, el canal sin la falla que hemos utilizado hasta ahora.

Consideraremos una variable externa *reset* que permitirá indicar la restauración física del enlance. Entonces, podemos modelar la falla de la siguiente manera:

```
FAULT enlace_roto
  pre(1);
  effect(arr = 0);
  restores(reset);
```

En nuestro modelo de canal, un proceso asigna 1 a la variable *arr* para notificar que hay un nuevo mensaje en el canal. Por lo tanto, para modelar la pérdida del mensaje, la falla debe hacer que la variable *arr* sea 0. El estado del sistema será restaurado cuando la variable *reset* sea 1.

El modelo del canal aumentado con la falla queda como se muestra a continuación:

```
MODULE Canal(reset)
FAULT enlace_roto
```

```
  pre(1);
  effect(arr = 0);
  restores(reset);
VAR
  arr : boolean;
  msg : 0..255;
INIT
  ! arr
```

Notemos que un canal con pérdida usual es el caso particular en que *reset* es igual a 1. Luego el modelo se simplifica eliminando la variable externa:

```
MODULE Canal
FAULT perdida
  pre(1);
  effect(arr = 0);
  restores(1);
VAR
  arr : boolean;
  msg : 0..255;
INIT
  ! arr
```

En el caso de ruido eléctrico, el mensaje podría dañarse y llegar alterado. Consideraremos que el valor 255 corresponde a un mensaje inválido. En este caso, el sistema se restaurará automáticamente al desaparecer el ruido. La falla se modela de la siguiente manera:

```
FAULT enlace_ruido
  pre(1);
  effect(msg = 255);
  restores(1);
```

A partir de los ejemplos vistos en este capítulo, el lector debería poder completar los modelos correspondientes al canal con pérdida y con ruido eléctrico.

7.8. Falla por reinicio

Este modelo de falla corresponde a un reinicio súbito del sistema. Por ejemplo, podría correspondar a una pérdida de energía en el equipo que ejecuta el proceso. En este caso, el efecto de la falla es volver todas las variables al valor que tenían en su estado inicial.

```
FAULT reinicio
  pre(1);
  effect(v_0 = y_0, ..., v_n = y_n);
  restores(1);
```

En nuestro modelo de ejemplo, asumimos que las variables del sistema son $v_0, \ldots, v_n$ y sus valores iniciales son $y_0, \ldots, y_n$.

7.9. Obseraciones finales

A lo largo de este capítulo, hemos presentado las fallas clásicas de la literatura y algunas otras que corresponden a casos particulares encontrados en sistemas reales. Además, hemos podido hacer un mapeo entre el modelo abstracto de la falla y su representación en el lenguaje de nuestra herramienta.

Capítulo 8

Casos de Estudio

En este capítulo, se presentan dos casos de estudio correspondientes a distintos problemas en sistemas distribuidos, podremos ver a FALLUTO en acción y comprobar su utilidad en el modelado de fallas. El primer caso es la resolución del problema de commit atómico (*atomic commit*) en una base de dato distribuida. El segundo corresponde a la elección de un proceso líder en una red con topología anular.

8.1. Caso 1: Commit atómico

El commit atómico (*atomic commit*) es un conjunto de cambios que deben ser realizados como una operación individual. En una base de datos esto se conoce como *transacción* y si alguno de los componentes no puede completarla, todos abortan la operación y vuelven atrás cualquier cambio intermedio.

8.1.1. Commit de dos fases

El algoritmo que presentamos [2] corresponde al protocolo de commit de dos fases (*two-phase commit*). Este algoritmo es un clásico en los sistemas de bases de datos distribuidos y puede encontrarse en varios productos tales como MySQL [38], Oracle [10] y DB2 [9].

En este sistema, los procesos pueden detenerse o reiniciarse (ver secciones 7.2 y 7.8, respectivamente). Para simplificar la exposición del modelo y sus fallas, sólo consideraremos el caso de los procesos que se detienen. La aplicación del modelo de fallas por reinicio es muy similar.

8.1.2. Descripción del algoritmo

Tal como lo sugiere el nombre, este protocolo consiste de dos fases. En la primera, cada proceso elige su voto y lo envía a un proceso distinguido c llamado "coordinador". En la segunda fase, el coordinador llega una decisión basándose en los votos recibidos y luego envía la decisión a los demás procesos.

Cada proceso vota por Sí o No y luego decide entre Commit o Abort, de modo tal que:

1. Si no ocurren fallas y todos los procesos votan por Sí, todos los procesos deciden hacer Commit.

2. Un proceso decide hacer Commit, sólo cuando todos los procesos votaron por Sí.

3. Todos los procesos que deciden, llegan a la misma decisión.

El proceso coordinador c tiene tres acciones. En la primera, c elige su voto, entra en la segunda fase y comienza a esperar los votos de los demás procesos. En la segunda acción, c detecta que todos los procesos han votado por Sí y decide hacer Commit. En la tercera acción, c detecta que algún proceso votó por No o se detuvo y decide hacer Abort.

Cada proceso j distinto del coordinador también tiene tres acciones. En la primera, j detecta que c ha votado y emite su voto. En la segunda acción, j detecta que c se detuvo y decide hacer Abort. En la tercera acción, j detecta que algún otro proceso ha completado su segunda fase y toma la misma decisión que éste.

Cada proceso tiene las siguientes variables:

- *phase* corresponderá a la fase del proceso; *phase* es 0 inicialmente, 1 cuando el proceso ya emitió su voto y 2 cuando ya decidió,

- d (según la fase actual del proceso) el voto o la decisión del proceso; $true$ si vota por Sí o la decisión es Commit y $false$ si vota por No o la decisión es Abort,

- *up* el estado actual del proceso; $true$ si está ejecutando y $false$ si se detuvo.

8.1.3. Modelo sin fallas

Consideremos el caso de cinco procesos: uno designado coordinador y cuatro no distinguidos. Cada proceso debe poder comunicarse con los demás para conocer sus fases y decisiones. Dado que no consideramos fallas en el envío o recepción de mensajes, podemos simplificar el modelo y permitir que un proceso conozca el valor de las variables de los demás. En FALLUTO esto se puede lograr pasando como parámetros de un proceso, la referencia a cada uno de los otros. Veamos entonces el modelo resultante para un proceso que no es el coordinador:

```
MODULE RegularVoter(c, v0, v1, v2)
VAR
  phase : {0, 1, 2};
  d     : boolean;
  up    : boolean;
INIT
  phase = 0 & up = 1;
ASSIGN
  next(phase) :=
    case
      phase = 0 & c.up & c.phase = 1    : 1;
      phase = 0 & !c.up                 : 2;
      phase < c.phase  & (c.phase = 2)  : 2;
      phase < v0.phase & (v0.phase = 2) : 2;
      phase < v1.phase & (v1.phase = 2) : 2;
      phase < v2.phase & (v2.phase = 2) : 2;
      1                                 : phase;
    esac;
  next(d) :=
    case
      phase = 0 & c.up & c.phase = 1    : {0, 1};
      phase = 0 & !c.up                 : 0;
      phase < c.phase  & (c.phase = 2)  : c.d;
      phase < v0.phase & (v0.phase = 2) : v0.d;
      phase < v1.phase & (v1.phase = 2) : v1.d;
      phase < v2.phase & (v2.phase = 2) : v2.d;
      1                                 : d;
    esac;
FAIRNESS running
```

En el estado inicial, las variables *phase* y *up* son 0 y 1 respectivamente. Nótese que la elección del voto d es no determinística en la primera fase cuando el coordinador ya votó (`phase = 0 & c.phase = 1`). Veamos ahora el modelo que corresponde al proceso coordinador:

```
MODULE Coordinator(v0, v1, v2, v3)
VAR
  phase : {0, 1, 2};
  d     : boolean;
  up    : boolean;
INIT
  phase = 0 & up = 1;
ASSIGN
  next(phase) :=
    case
      phase = 0 : 1;
      phase = 1
      & ( -- todos estan operativos y votaron si
            (v0.up & v0.phase = 1 & v0.d)
          & (v1.up & v1.phase = 1 & v1.d)
          & (v2.up & v2.phase = 1 & v2.d)
          & (v3.up & v3.phase = 1 & v3.d)
        ) : 2;
      phase = 1
      & ( -- algun proceso fallo o voto no
            (!v0.up | (v0.phase >= 1 & !v0.d))
          | (!v1.up | (v1.phase >= 1 & !v1.d))
          | (!v2.up | (v2.phase >= 1 & !v2.d))
          | (!v3.up | (v3.phase >= 1 & !v3.d))
        ) : 2;
      1   : phase;
    esac;

  next(d) :=
    case
      phase = 0 : {0, 1};
      phase = 1
      & ( -- todos estan operativos y votaron si
            (v0.up & v0.phase = 1 & v0.d)
          & (v1.up & v1.phase = 1 & v1.d)
          & (v2.up & v2.phase = 1 & v2.d)
```

```
          & (v3.up & v3.phase = 1 & v3.d)
        ) : 1;
      phase = 1
      & ( -- algun proceso fallo o voto no
            (!v0.up | (v0.phase >= 1 & !v0.d))
          | (!v1.up | (v1.phase >= 1 & !v1.d))
          | (!v2.up | (v2.phase >= 1 & !v2.d))
          | (!v3.up | (v3.phase >= 1 & !v3.d))
        ) : 0;
      1   : d;
    esac;
FAIRNESS running
```

Debería ser sencillo comprender las transiciones del coordinador a partir de la descripción del algoritmo presentada anteriormente. La especificación que nos interesa verificar nos dice que

- si el coordinador llegó a la fase 2 y decidió hacer Commit, entonces los demás procesos votaron que sí o hicieron Commit.
- si el coordinador está en la fase 0 o 1 (p. ej., sufrió una falla) o está en la fase 2 y decidió hacer Abort, entonces los demás procesos fallaron o hicieron Abort.

Es decir, la siguiente propiedad es un invariante del sistema:

$$c.phase = 2 \wedge c.d \Rightarrow (\forall j\ j.phase \neq 0 \wedge j.d)$$
$$\wedge\ (c.phase = 0 \vee c.phase = 1 \vee (c.phase = 2 \wedge \neg c.d))$$
$$\Rightarrow (\forall j\ j.phase \neq 2 \wedge \neg j.d)$$

Esta propiedad se satisface y estamos listos para aplicar el modelo de fallas según el proceso que presentamos en la sección 4.1.

8.1.4. Modelo con falla *fail-stop*

En nuestro sistema, la falla de tipo *fail-stop* sólo puede darse cuando la variable *up* es *true* y su ocurrencia inhabilita todas las transiciones del sistema. Por lo tanto, debemos declarar la falla según lo presentado en la sección 7.2 y agregar `disabled_by` en las asignaciones de las variables *phase* y *d*.

```
MODULE RegularVoter(c, v0, v1, v2)
FAULT FailStop
  pre(up);
  effect(up = 0);
  restores(0);
VAR
  phase : {0, 1, 2};
  d     : boolean;
  up    : boolean;
INIT
  phase = 0 & up = 1;
ASSIGN
  next(phase) :=
    case
      phase = 0 & c.up & c.phase = 1    : 1;
      phase = 0 & !c.up                 : 2;
      phase < c.phase  & (c.phase = 2)  : 2;
      phase < v0.phase & (v0.phase = 2) : 2;
      phase < v1.phase & (v1.phase = 2) : 2;
      phase < v2.phase & (v2.phase = 2) : 2;
      1                                 : phase;
    esac disabled_by {FailStop};
  next(d) :=
    case
      phase = 0 & c.up & c.phase = 1    : {0, 1};
      phase = 0 & !c.up                 : 0;
      phase < c.phase  & (c.phase = 2)  : c.d;
      phase < v0.phase & (v0.phase = 2) : v0.d;
      phase < v1.phase & (v1.phase = 2) : v1.d;
      phase < v2.phase & (v2.phase = 2) : v2.d;
      1                                 : d;
    esac disabled_by {FailStop};
FAIRNESS running
```

Nótese cuán sencillo es aplicar el modelo de esta falla en el programa. El modelo que corresponde al coordinador es similar al de los procesos no distinguidos. La propiedad también se satisface en el modelo con fallas. En el Apéndice D, encontramos el modelo completo correspondiente a este algoritmo con su falla *fail-stop* y su propiedad LTL.

8.2. Caso 2: Elección de líder

Este caso corresponde al problema de seleccionar un proceso como líder en un anillo sin procesos distinguidos. Por ejemplo, en una red de tipo "*token ring*", los distintos nodos hacen circular un "*token*" (un patrón especial de bits) dentro de la red. Algunas veces, este token puede perderse por alguna falla y debe ser regenerado. Durante el proceso de regeneración, los nodos eligen a uno para que actúe de líder y ponga a circular el token nuevamente.

8.2.1. Descripción del algoritmo

Consideremos un anillo undireccional con N procesos, $p_0, \ldots, p_{N-1}$, conectados en ese orden. Diremos que p_{i-1} es el *predecesor* del proceso p_i. Asumiremos que cada proceso tiene un identificador único y denotaremos con id_i al identificador del proceso p_i.

El algoritmo [26, 21, 40] elige el proceso con identificador mayor para que actúe como líder. Cada proceso p_i tiene dos variables: max_i, el mayor identificador que el proceso p_i conoce, y $dist_i$ la distancia al proceso con identificador max_i.

Cada proceso p_i tiene las siguientes tres acciones:

1. (El mayor identificador conocido es el propio.) Si id_i es mayor que max_i, max_i se hace id_i y $dist_i$ se hace 0. Si $dist_i$ es 0 y max_i es distinto a id_i, max_i se hace id_i. Si max_i es igual a id_i y $dist_i$ es distinto de 0, entonces $dist_i$ se hace 0.

2. (El predecesor conoce un *id* mayor al propio.) Si $dist_{i-1} + 1 < N$, o si id_i es menor que max_{i-1}, max_i se hace igual a max_{i-1} y $dist_i$ igual a $dist_{i-1} + 1$.

3. (El proceso se designa líder.) Si $dist_{i-1} + 1 \geq N$, o si id_i es mayor que id_{i-1} y id_i es mayor o igual a max_{i-1}, max_i se hace igual a id_i y $dist_i$ igual a 0.

Sea K el máximo identificador de los procesos en el anillo. Diremos que el sistema ha alcanzado su *estado estable* si:

1. Todos los procesos coinciden en quién es el líder.

   $max_i = K$, $i \in \{0, \ldots, N-1\}$

2. Todos los procesos conocen su distancia al líder.

   $id_j = K \Rightarrow dist_j = 0$ y para los procesos $i \neq j, dist_i = 1 + dist_{(i-1) mod N}$

El sistema está sujeto a fallas bizantinas que pueden alterar el valor de las variables de cada proceso (max_i y $dist_i$).

8.2.2. El modelo original

En la página 73, podemos ver el modelo correspondiente a cada proceso (*Host*). El argumento `host_id` es el identificador del proceso, `pred` es una referencia al proceso predecesor y `N`, el número total de procesos.

El modelo del sistema con cuatro procesos y su especificación pueden encontrarse en el módulo `main` de la página 74. Para favorecer la lectura, la definición de la macro `conocen_distancia` sólo contiene el caso de uno de los procesos; los otros tres son simétricos y pueden deducirse fácilmente.

La especificación LTL del sistema:

```
G F (mismo_lider & conocen_distancia)
```

nos dice que siempre en el futuro (**G**), en algún momento (**F**), vale que todos los procesos coinciden en quién es el líder (`mismo_lider`) y (`&`) conocen la distancia que los separa de éste (`conocen_distancia`).

El modelo satisface la especificación y, de acuerdo al proceso presentado en la sección 4.1, estamos en condiciones de aplicar el modelo de falla bizantina.

```
MODULE Host(host_id, pred, N)
DEFINE
  id := host_id;
VAR
  max  : 0 .. 3; -- debe entrar K
  dist : 0 .. 3; -- debe entrar N - 1
ASSIGN
 next(max) :=
  case
   (max < id) | (max != id & dist = 0) | (max = id & dist != 0) : id;
   (pred.dist + 1 < N) & (id < pred.max)                        : pred.max;
   ((pred.dist + 1 >= N) | (id > pred.id & id >= pred.max))     : id;
   1                                                            : max;
  esac;

 next(dist) :=
  case
   (max < id) | (max != id & dist = 0) | (max = id & dist != 0) : 0;
   (pred.dist + 1 < N) & (id < pred.max)                        : pred.dist + 1;
   ((pred.dist + 1 >= N) | (id > pred.id & id >= pred.max))     : 0;
   1                                                            : dist;
  esac;
FAIRNESS running
```

```
MODULE main
DEFINE
  N := 4; -- numero de nodos
  K := 3; -- maximo id de un host
VAR
  h0 : process Host(2, h3, N);
  h1 : process Host(1, h0, N);
  h2 : process Host(0, h1, N);
  h3 : process Host(3, h2, N);
FAIRNESS running

DEFINE mismo_lider := (h0.max = K & h1.max = K & h2.max = K & h3.max = K);
DEFINE conocen_distancia := (h0.id = K ->
                                h0.max = K & h0.dist = 0
                                & h1.max = K & h1.dist = 1
                                & h2.max = K & h2.dist = 2
                                & h3.max = K & h3.dist = 3)
                            --
                            -- ... resto de la definicion ...
                            -- ... simetrica para otros nodos ...
                            --

LTLSPEC G F (mismo_lider & conocen_distancia)
```

8.2.3. Aplicación de la falla

Como vimos en la sección 7.6, la falla bizantina puede ocurrir en cualquier momento, su efecto es asignar a cada variable algún valor de su dominio. y el sistema se restaura automáticamente. Por lo tanto, el modelo de falla es el que encontramos a continuación:

```
FAULT bizantina
  pre(1);
  effect(max = 0..3, dist = 0..3);
  restores(1);
```

Cuando introducimos esta falla, el modelo del sistema que presentamos en la sección anterior ya no satisface la especificación. Notemos que es posible tener ejecuciones donde las ocurrencias de la falla se alternan de modo tal que impiden a los procesos estabilizarse y elegir un líder.

8.2.4. Un modelo mejorado

En un sistema real, es razonable asumir que la ocurrencia de las fallas no estará constantemente impidiendo la estabilización de los procesos. Como NuSMV no permite el model checking probabilístico, debemos hacer alguna corrección en nuestro sistema que nos permita verificarlo. Una posibilidad, es calcular cuánto "tiempo" le toma al sistema estabilizarse y debilitar la especificación para que diga algo como "si durante tanto tiempo no hay fallas entonces el sistema puede elegir al líder". Dado que NuSMV tampoco admite model checking temporal, necesitamos recurrir a otro mecanismo para expresar la nueva especificación.

La solución que proponemos a este problema es agregar un contador al sistema que

- se incrementa en uno cada vez que un proceso se ejecuta, y
- vuelve a cero cuando algún proceso sufre la ocurrencia de una falla.

Sea `counter` la variable que representa al contador. Es fácil hacer que `counter` vuelva a cero cuando ocurre la falla: simplemente tenemos que agregarlo a la lista de efectos de la falla:

```
FAULT bizantina
  pre(1);
  effect(max = 0..3, dist = 0..3, counter = 0);
  restores(1);
```

El siguiente problema a resolver es cómo conseguimos que cada proceso incremente en uno el contador cada vez que se ejecuta. Para eso, agregamos la variable `token`. Cada vez que un proceso i incrementa en uno la variable `counter`, hace `token` igual a id_i. Un proceso sólo podrá incrementar la variable `counter` cuando su predecesor lo haya incrementado.

Tanto `counter` como `token` son variables del sistema que deben ser compartidas por todos los procesos. Por lo tanto, el módulo *Host* las recibirá como parámetros y tendrá estas nuevas acciones:

```
next(token) :=
  case
    token = pred.id : id;
    1               : token;
  esac;

next(counter) :=
  case
    token = pred.id & counter < MAX : counter + 1;
    1                               : counter;
  esac;
```

En un sistema con cuatro nodos, es suficiente que los procesos hayan podido ejecutarse diez veces para que el líder sea elegido. A continuación, puede verse la nueva especificación del sistema:

```
LTLSPEC G (
           (counter >= 10)
           ->
           (mismo_lider & conocen_distancia)
         )
```

8.3. Observaciones finales

En este capítulo mostramos casos reales donde aplicamos nuestro proceso de model checking y FALLUTO para verificar tolerancia a fallas. En el caso del algoritmo de la elección de líder, encontramos que el modelo original y su especificación eran inapropiados para verificar su tolerancia a fallas. Sin embargo, la herramienta nos ayudó a encontrar el error, corregirlo y luego verificar el sistema. En el Apéndice D pueden encontrarse los modelos completos correspondientes a estos casos de estudio.

Capítulo 9

Conclusión

¡Ya está despuntando er día!
¡Cantad, campanas, cantad!
¡que vuerve la gloria mía!

El amor brujo, Final., Manuel De Falla.
Libretto: Gregorio Martínez Sierra

Los principales aportes de este trabajo son la definición de un lenguaje declarativo de fallas muy sencillo, su implementación en una herramienta cuyo motor de model checking tiene calidad industrial y la definición de un proceso de verificación para sistemas tolerantes a fallas. La aplicación sistemática de este método en los casos de estudio posibilitó detectar errores a partir de los contraejemplos generados por FALLUTO.

El principal problema con que nos encontramos durante el desarrollo de este trabajo fue la alternancia equitativa entre los procesos y sus fallas. Si bien este problema pudo resolverse con la restricción de equidad fuerte en el modelo compilado, sería importante atacarlo con los comandos de equidad de NuSMV y estudiar el comportamiento de los nuevos modelos.

Hay algunos tipos de fallas que son muy similares en todos los modelos. Por ejemplo, las fallas bizantinas que pueden ocurrir en cualquier momento y de las cuales el sistema se recupera en el estado siguiente. Algo parecido ocurre con las fallas de crash, fail-stop o reinicio, que en todos los casos muestran una descripción semejante. Podría crearse una nueva sintaxis para ese tipo de fallas, haciendo opcionales la precondición y la condición de restauración. Por ejemplo, si tuviéramos una falla *"Crash"* predefinida, no sería necesario modificar todas las transiciones del sistema para que incluyan *"disabled_by {Crash}"*. Una falla *"Restart"* predefinida llevaría directamente el sistema a su estado inicial (declarado con INIT).

Además, FALLUTO podría incluir un conjunto de propiedades o esquemas de propiedades predefinidas que permitan hacer verificaciones generales sobre el sistema sin tener que escribir o pensar las fórmulas LTL. Por ejemplo, podríamos tener construcciones que verifiquen que "una falla sucede a lo sumo n veces".

Por último, sería interesante completar el parser de la herramienta para que entienda el lenguaje completo de NuSMV. De este modo, FALLUTO podría también funcionar como un entorno experimental para la exploración de nuevos tipos de fallas y su impacto en sistemas de tiempo real o para probar nuevas construcciones sintácticas del model checker (cuantificadores, asignación múltiple, etc).

Apéndice A

Parsing Expression Grammars

A.1. Introducción

Las gramáticas libres de contexto (CFG) y expresiones regulares (RE) se conocen como *sistemas generativos*: dan un conjunto de reglas que aplicadas recursivamente generan todas las palabras de un lenguaje. En contraposición, un *sistema de reconocimiento* define un lenguaje en término de reglas o predicados que deciden si una palabra dada pertenece al lenguaje.

Las *Parsing Expression Grammar* [16] (de ahora en más, PEG) son un sistema de reconocimiento sintáctico bastante similar a la Forma Normal de Backus-Naur (EBNF). La principal diferencia es que, en lugar del operador de elección '|' que indica alternativas sin orden para no terminales de EBNF, las PEG usan el operador de elección con prioridad '/'. De este modo, los patrones alternativos deben ser probados *en orden* incondicionalmente hasta la primer coincidencia. Por ejemplo, en EBNF las siguientes reglas son equivalentes:

$$A \rightarrow a \mid ab$$
$$A \rightarrow ab \mid a$$

Sin embargo, en PEG, las siguientes reglas son diferentes:

$$A \leftarrow a \,/\, ab$$
$$A \leftarrow ab \,/\, a$$

Ya que la cadena 'ab' no es reconocida por la primera regla, pero sí por la segunda.

Las PEG pueden ser comprendidas como la descripción formal de un *top-down parser*. Sin embargo, tienen más poder expresivo que la clase $LL(k)$ de

parsers, pueden expresar todos los parsers $LR(k)$ determinísticos e incluso algunas gramáticas sensibles al contexto. Un aspecto muy importante de las PEG es que pueden ser parseadas en tiempo lineal, pero con uso de memoria lineal con respecto al tamaño de la cadena por reconocer. En este apéndice, presentamos brevemente las PEG con una interpretación informal de su semántica y algunos resultados muy interesantes.

Definición 1. Una *PEG* es una 4-upla $G = (V_N, V_T, R, e_S)$, donde V_N es un conjunto finito de símbolos no terminales, V_T es un conjunto finito de símbolos terminales, e_S es una expresión de parsing llamada *expresión inicial*, y $V_N \cap V_T = \emptyset$. Cada regla $r \in R$ es un par (A, e), que lo escribimos $A \leftarrow e$ donde $A \in V_N$ y e es una expresión de parsing. Para cada no terminal A, hay exactamente una sola e tal que $A \leftarrow e \in R$. Por lo tanto, R es una función de no terminales en expresiones, y escribimos $R(A)$ para denotar la expresión e tal que $A \leftarrow e \in R$.

A continuación, damos una definición inductiva de las expresiones de parsing, junto con una noción intuitiva de su significado. Si e, e_1, e_2 expresiones de parsing, también lo son

ε, reconoce la cadena vacía.

a, con $a \in V_T$, que reconoce el símbolo terminal a.

A, con $A \in V_N$, que reconoce la regla asociada al símbolo no terminal A.

$e_1 e_2$, (secuencia) que reconoce e_1 y luego reconoce e_2.

e_1 / e_2, (elección con prioridad), reconoce e_1 y, si no puede, reconoce e_2.

$e*$, reconoce cero o más repeticiones de la expresión e.

$!e$, (predicado de negación) reconoce una cadena que no es reconocida por e sin mover el puntero de lectura.

., syntactic sugar equivalente a $a_1/a_2/\ldots/a_n$ con $a_i \in V_T$.

$e?$, syntactic sugar equivalente a e/ε.

$e+$, syntactic sugar equivalente a $ee*$.

$\&e$, syntactic sugar equivalente a $!(!e)$.

En 2004, Bryan Ford probó que para toda PEG G, es posible construir un parser de tiempo lineal que reconoce $L(G)$.

A.2. Treetop

Treetop [36] es una implementación de Parsing Expression Grammars en el lenguaje de programación Ruby [29]. Treetop provee un lenguaje específico para la declaración de la gramática y un compilador que convierte la gramática en un parser recursivo descendente. Los operadores de Treetop son iguales a los operadores presentados en la sección anterior.

La semántica correspondiente a las reglas de una gramática se puede dar como métodos de Ruby dentro de la definición de la regla o en un archivo separado. A continuación, se muestra un ejemplo sencillo de una gramática que reconoce palíndromos en el alfabeto $\{a, b\}$:

```
grammar Palindromos
  rule S
    'a' S 'a' / 'b' S 'b' / 'a' / 'b' / ''
  end
end
```

Apéndice B

Sintaxis formal de FALLUTO

En este apéndice, presentamos la gramática formal de FALLUTO expresada en PEG.

B.1. Producciones complejas

Un programa es una lista de uno o más módulos:

$$\begin{aligned} \textit{program} &\leftarrow \textit{module\_list} \\ \textit{module\_list} &\leftarrow \textit{module}+ \end{aligned}$$

Un módulo tiene un nombre, una signatura y una lista de cero o más declaraciones:

$$\begin{aligned} \textit{module} &\leftarrow \textit{MODULE module\_sign declaration}* \\ \textit{module\_sign} &\leftarrow \textit{ATOM (LP atom\_list RP)}? \\ \textit{atom\_list} &\leftarrow \textit{ATOM (COMMA ATOM)}* \end{aligned}$$

El nombre de un módulo, variable o constante (*ATOM*) es cualquier cadena alfanumérica que no sea una palabra reservada de FALLUTO. Para favorecer la explicación de la sintaxis, vamos a posponer momentáneamente la regla formal de *ATOM* y seguiremos con las declaraciones. Una declaración es el elemento básico de un módulo (declaración de variables, estado inicial, asignaciones, etc):

$$\begin{aligned} \textit{declaration} \leftarrow\ & \textit{fault} \,/\, \textit{var} \,/\, \textit{input\_var} \,/\, \textit{assign} \,/\, \textit{init} \,/\, \textit{invar} \,/ \\ & \textit{trans} \,/\, \textit{define} \,/\, \textit{fairness} \,/\, \textit{justice} \,/\, \textit{compassion} \,/ \\ & \textit{invarspec} \,/\, \textit{ctlspec} \,/\, \textit{ltlspec} \,/\, \textit{constants} \end{aligned}$$

Una falla tiene nombre, precondición, efecto y condición de restauración:

$$fault \leftarrow FAULT\ ATOM\ pre\ effect\ restores$$

La precondición de la falla es una expresión simple,

$$pre \leftarrow PRE\ LP\ simple\_expr\ RP\ SEMI$$

El efecto es una lista de cero o más variables variables con los valores que pueden tomar al ocurrir la falla.

$$\begin{aligned} effect &\leftarrow EFFECT\ LP\ effect\_expr\_list\ RP\ SEMI \\ effect\_expr\_list &\leftarrow effect\_expr?\ (COMMA\quad effect\_expr)* \\ effect\_expr &\leftarrow var\_id\ EQUAL\ simple\_expr \end{aligned}$$

La condición de restauración del sistema es una expresión sobre el próximo estado del sistema:

$$restores \leftarrow RESTORES\ LP\ next\_expr\ RP\ SEMI$$

La sección de declaración de variables y de *input vars* comienzan con *VAR* e *INVAR* respectivamente y luego una lista de cero o más variables con su nombre y un tipo:

$$\begin{aligned} var &\leftarrow VAR\ var\_decl* \\ var\_decl &\leftarrow decl\_var\_id\ COLON\ type\ SEMI \\ input\_var &\leftarrow IVAR\ ivar\_decl* \\ ivar\_decl &\leftarrow decl\_var\_id\ COLON\ type\ SEMI \end{aligned}$$

$$\begin{aligned} decl\_var\_id &\leftarrow ATOM\ decl\_var\_id\_tail \\ decl\_var\_id\_tail &\leftarrow DOT\ ATOM\ decl\_var\_id\_tail\ / \\ &\quad DOT\ NUMBER\ decl\_var\_id\_tail\ / \\ &\quad LB\ simple\_expr\ RB\ decl\_var\_id\_tail\ / \\ &\quad \varepsilon \end{aligned}$$

Éstas son las reglas para nombres de variables:

$$\begin{aligned} var\_id &\leftarrow ATOM\ var\_id\_tail\ /\ SELF\ var\_id\_tail \\ var\_id\_tail &\leftarrow DOT\ ATOM\ var\_id\_tail\ / \\ &\quad DOT\ NUMBER\ var\_id\_tail\ / \\ &\quad LB\ simple\_expr\ RB\ var\_id\_tail\ / \\ &\quad \varepsilon \end{aligned}$$

Los tipos de las variables pueden ser básicos (boolean, word, integer, etc), un módulo o un proceso:

$$
\begin{aligned}
\mathit{type} &\leftarrow \mathit{itype} \mid \mathit{module\_type} \mid \mathit{PROCESS}\ \mathit{module\_type} \\
\mathit{itype} &\leftarrow \mathit{BOOLEAN} \mid \mathit{subrange} \mid \\
&\quad \mathit{WORD}\ \mathit{LB}\ \mathit{simple\_expr}\ \mathit{RB} \mid \\
&\quad \mathit{INTEGER} \mid \mathit{LCB}\ \mathit{type\_value\_list}\ \mathit{RCB} \mid \\
&\quad \mathit{ARRAY}\ \mathit{WORD}\ \mathit{LB}\ \mathit{simple\_expr}\ \mathit{RB}\ \mathit{OF} \\
&\quad\quad \mathit{WORD}\ \mathit{LB}\ \mathit{simple\_expr}\ \mathit{RB} \mid \\
&\quad \mathit{ARRAY}\ \mathit{subrange}\ \mathit{OF}\ \mathit{itype} \\
\mathit{type\_value\_list} &\leftarrow \mathit{type\_value}\ (\mathit{COMMA}\quad \mathit{type\_value})* \\
\mathit{type\_value} &\leftarrow \mathit{complex\_atom} \mid \mathit{integer} \mid \mathit{FALSE} \mid \mathit{TRUE} \\
\mathit{module\_type} &\leftarrow \mathit{ATOM}\ \mathit{LP}\ \mathit{simple\_list\_expr}\ \mathit{RP} \mid \\
&\quad \mathit{ATOM}\ \mathit{LP}\ \mathit{RP} \mid \\
&\quad \mathit{ATOM} \mid \\
&\quad \mathit{ARRAY}\ \mathit{subrange}\ \mathit{OF}\ \mathit{module\_type} \\
\mathit{simple\_list\_expr} &\leftarrow \mathit{simple\_expr}\ (\mathit{COMMA}\quad \mathit{simple\_expr})*
\end{aligned}
$$

La sección de constantes es una lista de una o más declaraciones de constantes:

$$
\begin{aligned}
\mathit{constants} &\leftarrow \mathit{CONSTANTS}\ \mathit{const\_expr}+\ \mathit{SEMI}? \\
\mathit{const\_expr} &\leftarrow \mathit{complex\_atom}\ (\mathit{COMMA}\quad \mathit{complex\_atom})* \\
\mathit{complex\_atom} &\leftarrow \mathit{ATOM}\ (\mathit{DOT}\quad \mathit{ATOM})*
\end{aligned}
$$

La sección de macros se declara con *DEFINE* seguido de una o más declaraciones de macros:

$$
\begin{aligned}
\mathit{define} &\leftarrow \mathit{DEFINE}\ \mathit{define\_decl}+ \\
\mathit{define\_decl} &\leftarrow \mathit{decl\_var\_id}\ \mathit{EQDEF}\ \ \mathit{simple\_expr}\ \mathit{SEMI}
\end{aligned}
$$

La sección de asignaciones comienza con *ASSIGN* y una lista de una o más asignaciones de variables. La regla original de asignación de variables de NuSMV fue modificada para contemplar la inhabilitación de transiciones por fallas. Recordemos que una transición puede estar inhabilitada por una

lista de fallas entre llaves. Las reglas sintácticas son las siguientes:

$$
\begin{aligned}
\mathit{assign} &\leftarrow \mathit{ASSIGN\ one\_assign}+ \\
\mathit{one\_assign} &\leftarrow \mathit{var\_id\ EQDEF\ simple\_expr\ SEMI}\ / \\
&\quad \mathit{SMALLINIT\ LP\ var\_id\ RP\ EQDEF\ simple\_expr\ SEMI}\ / \\
&\quad \mathit{NEXT\ LP\ var\_id\ RP\ EQDEF\ basic\_expr} \\
&\quad\quad \mathit{DISABLED\_BY\ LCB\ fault\_list\ RCB\ SEMI}\ / \\
&\quad \mathit{NEXT\ LP\ var\_id\ RP\ EQDEF\ basic\_expr\ SEMI} \\
\mathit{fault\_list} &\leftarrow \mathit{ATOM}\ (\mathit{COMMA\ ATOM})*
\end{aligned}
$$

La regla para describir el estado inicial:

$$\mathit{init} \leftarrow \mathit{INIT\ simple\_expr\ SEMI}?$$

La regla para describir invariantes:

$$\mathit{invar} \leftarrow \mathit{INVAR\ simple\_expr\ SEMI}?$$

La regla para describir transiciones como fórmulas proposicionales en vez usar *ASSIGN*:

$$\mathit{trans} \leftarrow \mathit{TRANS\ simple\_expr\ SEMI}?$$

Las reglas para las condiciones de equidad:

$$
\begin{aligned}
\mathit{fairness} &\leftarrow \mathit{FAIRNESS\ simple\_expr\ SEMI}? \\
\mathit{justice} &\leftarrow \mathit{JUSTICE\ simple\_expr\ SEMI}? \\
\mathit{compassion} &\leftarrow \mathit{COMPASSION\ LP\ simple\_expr\ COMMA} \\
&\quad\quad \mathit{simple\_expr\ RP\ SEMI}?
\end{aligned}
$$

La regla para describir especificaciones de invariantes:

$$\mathit{invarspec} \leftarrow \mathit{INVARSPEC\ simple\_expr\ SEMI}?$$

La regla para especifiaciones LTL:

$$\mathit{ltlspec} \leftarrow \mathit{LTLSPEC\ ltl\_expression\ SEMI}?$$

La regla para especifiaciones CTL:

$$\mathit{ctlspec} \leftarrow \mathit{SPEC\ ctl\_expression\ SEMI}?\ /\ \mathit{CTLSPEC\ ctl\_expression\ SEMI}?$$

Las expresiones LTL y CTL (*ltl_expression* y *ctl_expression*, respectivamente), las expresiones sobre el estado actual y el estado siguiente (*simple_expr* y *next_expr*, respectivamente) son todas una expresión básica (*basic_expr*):

$$
\begin{aligned}
\mathit{ctl\_expression} &\leftarrow \mathit{basic\_expr} \\
\mathit{ltl\_expression} &\leftarrow \mathit{basic\_expr} \\
\mathit{simple\_expr} &\leftarrow \mathit{basic\_expr} \\
\mathit{next\_expr} &\leftarrow \mathit{basic\_expr}
\end{aligned}
$$

Las reglas que se muestran a continuación, corresponden a una expresión básica:

$$
\begin{aligned}
\mathit{basic\_expr} &\leftarrow \mathit{implies\_expr} \\
\mathit{implies\_expr} &\leftarrow \mathit{iff\_expr}\ (\textit{IMPLIES}\ \mathit{iff\_expr})* \\
\mathit{iff\_expr} &\leftarrow \mathit{or\_expr}\ (\textit{IFF}\ \mathit{or\_expr})* \\
\mathit{or\_expr} &\leftarrow \mathit{and\_expr}\ \mathit{or\_expr\_tail} \\
\mathit{or\_expr\_tail} &\leftarrow \textit{OR}\ \mathit{and\_expr}\ \mathit{or\_expr\_tail}\ / \\
&\qquad \textit{XOR}\ \mathit{and\_expr}\ \mathit{or\_expr\_tail}\ / \\
&\qquad \textit{XNOR}\ \mathit{and\_expr}\ \mathit{or\_expr\_tail}\ / \\
&\qquad \textit{EPSILON} \\
\mathit{and\_expr} &\leftarrow \mathit{ltl\_binary\_expr}\ (\textit{AND}\ \mathit{ltl\_binary\_expr})*
\end{aligned}
$$

Como se puede notar, las expresiones básicas se reducen a expresiones LTL binarias, y éstas, a su vez, en expresiones CTL simples:

$$
\begin{aligned}
\mathit{ltl\_binary\_expr} &\leftarrow \mathit{ltl\_unary\_expr}\ \mathit{ltl\_binary\_expr\_tail} \\
\mathit{ltl\_binary\_expr\_tail} &\leftarrow \textit{UNTIL}\ \mathit{ltl\_unary\_expr}\ \mathit{ltl\_binary\_expr\_tail}\ / \\
&\qquad \textit{SINCE}\ \mathit{ltl\_unary\_expr} \\
&\qquad\quad \mathit{ltl\_binary\_expr\_tail}\ / \\
&\qquad \textit{RELEASES}\ \mathit{ltl\_unary\_expr} \\
&\qquad\quad \mathit{ltl\_binary\_expr\_tail}\ / \\
&\qquad \textit{TRIGGERED}\ \mathit{ltl\_unary\_expr} \\
&\qquad\quad \mathit{ltl\_binary\_expr\_tail}\ / \\
&\qquad \textit{EPSILON} \\
\mathit{ltl\_unary\_expr} &\leftarrow \mathit{ctl\_expr}\ \mathit{ltl\_unary\_expr}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{pure\_ltl\_unary\_expr} \leftarrow\ & \mathit{OP\_NEXT\ ltl\_unary\_expr} \\
& \mathit{OP\_PREC\ ltl\_unary\_expr} \\
& \mathit{OP\_NOTPRECNOT\ ltl\_unary\_expr} \\
& \mathit{OP\_GLOBAL\ ltl\_unary\_expr} \\
& \mathit{OP\_HISTORICAL\ ltl\_unary\_expr} \\
& \mathit{OP\_FUTURE\ ltl\_unary\_expr} \\
& \mathit{OP\_ONCE\ ltl\_unary\_expr} \\
& \mathit{OP\_NOT\ pure\_ltl\_unary\_expr}
\end{aligned}
$$

Las expresiones CTL simples se componen de expresiones relacionales:

$$
\begin{aligned}
\mathit{ctl\_expr} \leftarrow\ & \mathit{relational\_expr} \ /\ \mathit{pure\_ctl\_expr} \\
\mathit{pure\_ctl\_expr} \leftarrow\ & \mathit{EX\ ctl\_expr} \ / \\
& \mathit{AX\ ctl\_expr} \ / \\
& \mathit{EF\ ctl\_expr} \ / \\
& \mathit{AF\ ctl\_expr} \ / \\
& \mathit{EG\ ctl\_expr} \ / \\
& \mathit{AG\ ctl\_expr} \ / \\
& \mathit{AA\ LB\ ctl\_basic\_expr} \\
& \quad \mathit{UNTIL\ ctl\_basic\_expr\ RB} \ / \\
& \mathit{EE\ LB\ ctl\_basic\_expr} \\
& \quad \mathit{UNTIL\ ctl\_basic\_expr\ RB} \ / \\
& \mathit{AA\ LB\ ctl\_basic\_expr} \\
& \quad \mathit{BUNTIL\ ctl\_basic\_expr\ RB} \ / \\
& \mathit{EE\ LB\ ctl\_basic\_expr} \\
& \quad \mathit{BUNTIL\ ctl\_basic\_expr\ RB} \ / \\
& \mathit{EBF\ subrange\ ctl\_expr} \ / \\
& \mathit{ABF\ subrange\ \_ctl\_expr} \ / \\
& \mathit{EBG\ subrange\ \_ctl\_expr} \ / \\
& \mathit{ABG\ subrange\ \_ctl\_expr} \ / \\
& \mathit{NOT\ pure\_ctl\_expr}
\end{aligned}
$$

$$
\begin{aligned}
ctl\_basic\_expr &\leftarrow ctl\_implies\_expr \\
ctl\_implies\_expr &\leftarrow ctl\_iff\_expr\ IMPLIES\ ctl\_implies\_expr\ / \\
&\quad\ ctl\_iff\_expr \\
ctl\_iff\_expr &\leftarrow ctl\_or\_expr\ (IFF\ ctl\_or\_expr)* \\
ctl\_or\_expr &\leftarrow ctl\_and\_expr\ ctl\_or\_expr\_tail \\
ctl\_or\_expr\_tail &\leftarrow OR\ ctl\_and\_expr\ ctl\_or\_expr\_tail\ / \\
&\quad\ XOR\ ctl\_and\_expr\ ctl\_or\_expr\_tail\ / \\
&\quad\ XNOR\ ctl\_and\_expr\ ctl\_or\_expr\_tail\ / \\
&\quad\ EPSILON \\
ctl\_and\_expr &\leftarrow ctl\_expr\ (AND\ ctl\_expr)*
\end{aligned}
$$

Las expresiones proposicionales se descomponen en operaciones primarias (operaciones sobre conjuntos, bits, lógicas y aritméticas):

$$
\begin{aligned}
relational\_expr &\leftarrow in\_expr\ relational\_expr\_tail \\
relational\_expr\_tail &\leftarrow EQUAL\ in\_expr\ relational\_expr\_tail\ / \\
&\quad\ NOTEQUAL\ in\_expr\ relational\_expr\_tail\ / \\
&\quad\ GT\ in\_expr\ relational\_expr\_tail\ / \\
&\quad\ GE\ in\_expr\ relational\_expr\_tail\ / \\
&\quad\ LT\ in\_expr\ relational\_expr\_tail\ / \\
&\quad\ LE\ in\_expr\ relational\_expr\_tail\ / \\
&\quad\ EPSILON \\
in\_expr &\leftarrow union\_expr\ (SETIN\ union\_expr)* \\
union\_expr &\leftarrow set\_expr\ (UNION\ set\_expr)* \\
set\_expr &\leftarrow subrange\ /\ shift\_expr\ /\ LCB\ set\_list\_expr\ RCB \\
set\_list\_expr &\leftarrow basic\_expr\ (COMMA\ basic\_expr)* \\
shift\_expr &\leftarrow remainder\_expr\ shift\_expr\_tail \\
shift\_expr\_tail &\leftarrow LSHIFT\ remainder\_expr\ shift\_expr\_tail\ / \\
&\quad\ RSHIFT\ remainder\_expr\ shift\_expr\_tail\ / \\
&\quad\ EPSILON
\end{aligned}
$$

remainder_ expr ← *additive_ expr* (*MOD additive_ expr*)∗
additive_ expr ← *multiplicative_ expr additive_ expr_ tail*
additive_ expr_ tail ← *PLUS multip_ expr additive_ expr_ tail* /
MINUS multip_ expr additive_ expr_ tail /
EPSILON
multip_ expr ← *concat_ expr multip_ expr_ tail*
multip_ expr_ tail ← *TIMES multip_ expr concat_ expr_ tail* /
DIVIDE multip_ expr concat_ expr_ tail /
EPSILON
concat_ expr ← *primary_ expr*
(*CONCATENATION primary_ exp*)∗
primary_ exp ← *constant primary_ exp_ tail* /
MINUS primary_ exp primary_ exp_ tail /
ATOM primary_ exp_ tail /
SELF primary_ exp_ tail /
LP basic_ exp RP primary_ exp_ tail /
NOT primary_ exp primary_ exp_ tail /
BOOL LP basic_ exp RP primary_ exp_ tail /
WORD1 LP basic_ exp RP primary_ exp_ tail /
NEXT LP basic_ exp RP primary_ exp_ tail /
CASE (*case_ element_ expr*) + *ESAC*
primary_ exp_ tail
primary_ exp_ tail ← *DOT ATOM primary_ exp_ tail* /
DOT NUMBER primary_ exp_ tail /
LB simple_ expression RB primary_ exp_ tail /
LB simple_ expression RB simple_ expression
COLON primary_ exp_ tail /
EPSILON
case_ element_ expr ← *basic_ expr COLON basic_ expr*
(*DISABLED_ BY LCB fault_ list RCB*)? *SEMI*

$$
\begin{aligned}
\mathit{constant} &\leftarrow \mathit{FALSEEXP} \;/\; \mathit{TRUEEXP} \;/\; \mathit{number\_word} \;/\; \mathit{number} \\
\mathit{number} &\leftarrow \mathit{NUMBER} \;/\; \mathit{PLUS\ NUMBER} \\
\mathit{integer} &\leftarrow \mathit{NUMBER} \;/\; \mathit{PLUS\ NUMBER} \;/\; \mathit{MINUS\ NUMBER} \\
\mathit{number\_word} &\leftarrow \mathit{NUMBER\_WORD} \\
\mathit{subrange} &\leftarrow \mathit{integer\ TWODOTS\ integer}
\end{aligned}
$$

B.2. Producciones simples

Todas las reglas para los operadores consumen los caracteres en blanco (sp), y las palabras claves consumen el fin de una palabra clave (eok) y los caracteres en blanco.

$$
\begin{aligned}
\mathit{AND} &\leftarrow \text{`\&'}\ sp \\
\mathit{COLON} &\leftarrow \text{`:'}\ sp \\
\mathit{COMMA} &\leftarrow \text{`,'}\ sp \\
\mathit{DOT} &\leftarrow \text{`.'}\ sp \\
\mathit{CONCATENATION} &\leftarrow \text{`::'}\ sp \\
\mathit{DIVIDE} &\leftarrow \text{`/'}\ sp \\
\mathit{EQDEF} &\leftarrow \text{`:='}\ sp \\
\mathit{EQUAL} &\leftarrow \text{`='}\ sp \\
\mathit{GE} &\leftarrow \text{`>='}\ sp \\
\mathit{GT} &\leftarrow \text{`>'}\ sp \\
\mathit{IFF} &\leftarrow \text{`<->'}\ sp \\
\mathit{IMPLIES} &\leftarrow \text{`->'}\ sp \\
\mathit{LB} &\leftarrow \text{`[']}\ sp \\
\mathit{LCB} &\leftarrow \text{`\{'}\ sp \\
\mathit{LE} &\leftarrow \text{`<='}\ sp \\
\mathit{LP} &\leftarrow \text{`('}\ sp \\
\mathit{LSHIFT} &\leftarrow \text{`<<'}\ sp \\
\mathit{LT} &\leftarrow \text{`<'}\ sp \\
\mathit{MINUS} &\leftarrow \text{`-'}\ sp
\end{aligned}
$$

$$
\begin{aligned}
MOD &\leftarrow \text{`mod' } eok\ sp \\
NOT &\leftarrow \text{`!' } sp \\
NOTEQUAL &\leftarrow \text{`!=' } sp \\
OR &\leftarrow \text{`|' } sp \\
PLUS &\leftarrow \text{`+' } sp \\
RB &\leftarrow \text{`]' } sp \\
RCB &\leftarrow \text{`}' } sp \\
RP &\leftarrow \text{`)' } sp \\
RSHIFT &\leftarrow \text{`>>' } sp \\
SEMI &\leftarrow \text{`;' } sp \\
TWODOTS &\leftarrow \text{`..' } sp \\
TIMES &\leftarrow \text{`*' } sp \\
XNOR &\leftarrow \text{`xnor' } sp \\
XOR &\leftarrow \text{`xor' } sp \\
AA &\leftarrow \text{`A' } eok\ sp \\
ABF &\leftarrow \text{`ABF' } eok\ sp \\
ABG &\leftarrow \text{`ABG' } eok\ sp \\
AF &\leftarrow \text{`AF' } eok\ sp \\
AG &\leftarrow \text{`AG' } eok\ sp \\
ARRAY &\leftarrow \text{`array' } eok\ sp \\
ASSIGN &\leftarrow \text{`ASSIGN' } eok\ sp \\
AX &\leftarrow \text{`AX' } eok\ sp \\
BOOLEAN &\leftarrow \text{`boolean' } eok\ sp \\
BOOL &\leftarrow \text{`bool' } eok\ sp \\
BUNTIL &\leftarrow \text{`BU' } eok\ sp \\
CASE &\leftarrow \text{`case' } eok\ sp \\
COMPASSION &\leftarrow \text{`COMPASSION' } eok\ sp \\
COMPUTE &\leftarrow \text{`COMPUTE' } eok\ sp \\
CONSTANTS &\leftarrow eok\ sp \\
CTLSPEC &\leftarrow eok\ sp \\
DEFINE &\leftarrow \text{`DEFINE' } eok\ sp \\
EBF &\leftarrow \text{`EBF' } eok\ sp \\
EBG &\leftarrow \text{`EBG' } eok\ sp \\
EE &\leftarrow \text{`EE' } eok\ sp \\
EF &\leftarrow \text{`EF' } eok\ sp \\
EG &\leftarrow \text{`EG' } eok\ sp
\end{aligned}
$$

$$
\begin{aligned}
ESAC &\leftarrow \text{`esac' eok sp} \\
EX &\leftarrow \text{`EX' eok sp} \\
FAIRNESS &\leftarrow \text{`FAIRNESS' eok sp} \\
FALSEEXP &\leftarrow \text{`FALSE' eok sp} \\
INIT &\leftarrow \text{`INIT' eok sp} \\
INTEGER &\leftarrow \text{`integer' eok sp} \\
INVAR &\leftarrow \text{`INVAR' eok sp} \\
INVARSPEC &\leftarrow \text{`INVARSPEC' eok sp} \\
ISA &\leftarrow \text{`ISA' eok sp} \\
IVAR &\leftarrow \text{`IVAR' eok sp} \\
JUSTICE &\leftarrow \text{`JUSTICE' eok sp} \\
LTLSPEC &\leftarrow \text{`LTLSPEC' eok sp} \\
MMAX &\leftarrow \text{`MAX' eok sp} \\
MMIN &\leftarrow \text{`MIN' eok sp} \\
MODULE &\leftarrow \text{`MODULE' eok sp} \\
NEXT &\leftarrow \text{`next' eok sp} \\
OF &\leftarrow \text{`of' eok sp} \\
OP\_FUTURE &\leftarrow \text{`F' eok sp} \\
OP\_GLOBAL &\leftarrow \text{`G' eok sp} \\
OP\_HISTORICAL &\leftarrow \text{`H' eok sp} \\
OP\_NEXT &\leftarrow \text{`X' eok sp} \\
OP\_NOTPRECNOT &\leftarrow \text{`Z' eok sp} \\
OP\_ONCE &\leftarrow \text{`O' eok sp} \\
OP\_PREC &\leftarrow \text{`Y' eok sp} \\
PROCESS &\leftarrow \text{`process' eok sp} \\
RELEASES &\leftarrow \text{`V' eok sp} \\
SELF &\leftarrow \text{`self' eok sp} \\
SETIN &\leftarrow \text{`in' eok sp} \\
SINCE &\leftarrow \text{`S' eok sp} \\
SMALLINIT &\leftarrow \text{`init' eok sp} \\
SPEC &\leftarrow \text{`SPEC' eok sp} \\
TRANS &\leftarrow \text{`TRANS' eok sp} \\
TRIGGERED &\leftarrow \text{`T' eok sp} \\
TRUEEXP &\leftarrow \text{`TRUE' eok sp} \\
UNION &\leftarrow \text{`union' eok sp}
\end{aligned}
$$

$$
\begin{aligned}
UNTIL &\leftarrow \text{`U' eok sp} \\
VAR &\leftarrow \text{`VAR' eok sp} \\
WORD1 &\leftarrow \text{`word1' eok sp} \\
WORD &\leftarrow \text{`word' eok sp} \\
FAULT &\leftarrow \text{`FAULT' eok sp} \\
PRE &\leftarrow \text{`pre' eok sp} \\
EFFECT &\leftarrow \text{`effect' eok sp} \\
RESTORES &\leftarrow \text{`restores' eok sp} \\
DISABLED\_BY &\leftarrow \text{`disabled\_by' eok sp} \\
NUMBER\_WORD &\leftarrow \text{`0' [bBoOdDhH] [0-9]* `\_'} \\
&\qquad \text{[0-9a-fA-F] [0-9a-fA-F\_]*}
\end{aligned}
$$

La regla keyword coincide con cualquier palabra clave del lenguaje y se llega al final de una palabra (*eok*) reservada cuando se encuentra un símbolo que no puede formarla (*idsyms*):

$$
\begin{aligned}
keyword \leftarrow\ & AA \,/\, ABF \,/\, ABG \,/\, AF \,/\, AG \,/\, ARRAY \,/\, ASSIGN \,/ \\
& AX \,/\, BOOLEAN \,/\, BOOL \,/\, BUNTIL \,/\, CASE \,/ \\
& COMPASSION \,/\, COMPUTE \,/\, CONSTANTS \,/ \\
& CTLSPEC \,/\, DEFINE \,/\, EBF \,/\, EBG \,/\, EE \,/\, EF \,/ \\
& EG \,/\, ESAC \,/\, EX \,/\, FAIRNESS \,/\, FALSEEXP \,/ \\
& INIT \,/\, INTEGER \,/\, INVARSPEC \,/\, INVAR \,/\, ISA \,/ \\
& IVAR \,/\, JUSTICE \,/\, LTLSPEC \,/\, MMAX \,/\, MMIN \,/ \\
& MODULE \,/\, NEXT \,/\, OF \,/\, OP\_FUTURE \,/ \\
& OP\_GLOBAL \,/\, OP\_HISTORICAL \,/\, OP\_NEXT \,/ \\
& OP\_NOTPRECNOT \,/\, OP\_ONCE \,/\, OP\_PREC \,/ \\
& PROCESS \,/\, RELEASES \,/\, SELF \,/\, SETIN \,/\, SINCE \,/ \\
& SMALLINIT \,/\, SPEC \,/\, TRANS \,/\, TRIGGERED \,/ \\
& TRUEEXP \,/\, UNION \,/\, UNTIL \,/\, VAR \,/\, WORD1 \,/ \\
& WORD \,/\, FAULT \,/\, PRE \,/\, EFFECT \,/\, RESTORES \,/ \\
& DISABLED\_BY \\
eok \leftarrow\ & ! \; idsyms \\
idsyms \leftarrow\ & [A\text{-}Za\text{-}z0\text{-}9\_\$\#]
\end{aligned}
$$

Las reglas para números y átomos son muy sencillas:

NUMBER ← *[0-9]+ sp*
ATOM ← *! keyword [A-Za-z_] [A-Za-z0-9_ $#]* sp*

La regla para el espacio en blanco consume caracteres blancos o comentarios hasta el fin de un comentario (*eoc*) o fin del archivo (*eof*):

sp ← *(white / comment)**
white ← *[\r\n\t]*
comment ← *'- -' (!eoc .)* eoc*
eoc ← *"\n" / eof*
eof ← *! .*

Por último, la regla que coincide con la cadena vacía:

EPSILON ← *''*

Apéndice C

Manual de FALLUTO

En este apéndice, se describen todas las herramientas que fueron desarrolladas para este trabajo final y se muestran las opciones con las cuales se puede modificar el comportamiento por defecto.

FALLUTO

FALLUTO funciona como un *frontend* para la verificación del modelo. El nombre del ejecutable es `falluto.rb` y sus opciones son:

-f, –file <model_file> Indica el archivo *input_file* que contiene el modelo a verificar.

-g, –graph <pdf_file> Indica el archivo *pdf_file* que contendrá el gráfico del autómata correspondiente al contraejemplo de la propiedad.

-n, –nusmv <opciones> Pasa *opciones* a NuSMV al verificar el modelo.

-h, –help Imprime la ayuda de FALLUTO y termina.

-v, –version Imprime el número de versión de FALLUTO y termina.

Con sólo especificar el modelo de entrada, FALLUTO compila, verifica el modelo y muestra una traza de contraejemplo si no se cumple la propiedad verificada. La generación del autómata con el contraejemplo es opcional al igual que los parámetros adicionales para NuSMV.

Compilador

El compilador convierte el código escrito en el lenguaje de FALLUTO a código de NuSMV. El nombre del ejecutable es `ftnusmv.rb` y sus opciones son:

-f, –file <model_file> Indica el archivo de entrada *trace_file* que contiene el modelo a compilar.

-a, –auxfile <aux_file> Indica el archivo de salida *aux_file* que contendrá los objetos de soporte de la compilación.

-o, –output <nusmv_model> Indica el archivo de salida *nusmv_model* que contendrá el modelo compilado.

-h, –help Imprime la ayuda del compilador y termina.

-v, –version Imprime el número de versión del compilador y termina.

Intérprete de traza

El intérprete de traza limpia la salida de NuSMV y quita cualquier variable auxiliar u otros objetos de soporte creados por el compilador para modelar las fallas. El nombre del ejecutable es `cleantrace.rb` y sus opciones son:

-f, –file <trace_file> Indica el archivo de entrada *trace_file* que contiene la traza por limpiar.

-a, –auxfile <aux_file> Indica el archivo de entrada *aux_file* que contiene los objetos de soporte creados por el compilador.

-o, –output <clean_trace> Indica el archivo de salida *clean_trace* que contendrá la traza limpia.

-h, –help Imprime la ayuda del intérprete de traza y termina.

-v, –version Imprime el número de versión del intérprete de traza y termina.

Visualizador de traza

El visualizador de traza genera el gráfico del autómata correspondiente al contraejemplo encontrado por NuSMV en un archivo con formato PDF. El nombre del ejecutable es `viewtrace.rb` y sus opciones son:

-f, –file <trace_file> Indica el archivo de entrada *trace_file* que contiene la traza a partir de la cual se genera el autómata de contraejemplo.

-d, –dotfile <dot_file> Indica el archivo de salida *dot_file* que contendrá el autómata de contraejemplo en formato dot.

-o, –output <pdf_file> Indica el archivo de salida *pdf_file* que contendrá el autómata de contraejemplo en formato PDF.

-h, –help Imprime la ayuda del visualizador de traza y termina.

-v, –version Imprime el número de versión del visualizador de traza y termina.

Ejemplo de ejecución

En esta sección, mostramos una ejecución de ejemplo de la verificación del modelo contenido en `/tmp/leader_fault.smv`.

```
$ ./falluto.rb -f /tmp/leader_fault.smv  -g
FALLUTO version 0.1.0

# Compiling...
./ftnusmv.rb -f /tmp/leader_fault.smv -o leader_fault.out
    -a leader_fault.aux
Reading input model from: /tmp/leader_fault.smv
Writing compiled model to: leader_fault.out
Writing support objects to: leader_fault.aux
Done.
# Model checking...
NuSMV  leader_fault.out > leader_fault.trace
# Cleaning trace...
./cleantrace.rb -f leader_fault.trace -a leader_fault.aux
    -o leader_fault.clean
Reading NuSMV trace from: leader_fault.trace
Reading support objects from: leader_fault.aux
Writing clean trace to: leader_fault.clean
Done.
# Graphing automata...
./viewtrace.rb -f leader_fault.trace -d leader_fault.dot
    -o leader_fault.pdf
Reading clean trace from: leader_fault.trace
Writing dot file to: leader_fault.dot
Writing automata to: leader_fault.pdf
Done.
```

Apéndice D

Casos de estudio completos

En este apéndice, se puede encontrar los modelos completos correspondientes a los casos de estudio del capítulo 8. En la primera parte, se presenta el caso del commit atómico. En la segunda, el problema de la elección de líder resuelto con el contador y token que modelan el paso del tiempo.

Listing D.1: Caso 1: Commit Atómico (Parte 1 / 5)

```
MODULE Coordinator(v0, v1, v2, v3)
FAULT crash
  pre(1);
  effect(up = 0);
  restores(0);
VAR
  phase : 0..2;
  d : boolean;
  up : boolean;
INIT
  phase = 0 &  up;
ASSIGN
  next(phase) :=
    case
      phase = 0 : 1;
      phase = 1 & ( -- all processes are up and voted yes
        (v0.up & v0.phase = 1 & v0.d) & (v1.up & v1.phase = 1 & v1.d)
        & (v2.up & v2.phase = 1 & v2.d) & (v3.up & v3.phase = 1 & v3.d)
        ) : 2;
      phase = 1 & ( -- some process crashed or voted 'no'
        (!v0.up | (v0.phase >= 1 & !v0.d)) | (!v1.up | (v1.phase >= 1 & !v1.d))
        | (!v2.up | (v2.phase >= 1 & !v2.d)) | (!v3.up | (v3.phase >= 1 & !v3.d))
        ) : 2;
      1 : phase;
    esac disabled_by {crash};
```

Listing D.2: Caso 1: Commit Atómico (Parte 2 / 5)

```
  next(d) :=
    case
      phase = 0 : {0, 1};
      phase = 1 & ( -- all processes are up and voted yes
        (v0.up & v0.phase = 1 & v0.d) & (v1.up & v1.phase = 1 & v1.d)
        & (v2.up & v2.phase = 1 & v2.d) & (v3.up & v3.phase = 1 & v3.d)
        ) : 1;
      phase = 1 & ( -- some process crashed or voted 'no'
        (!v0.up | (v0.phase >= 1 & !v0.d)) | (!v1.up | (v1.phase >= 1 & !v1.d))
        | (!v2.up | (v2.phase >= 1 & !v2.d)) | (!v3.up | (v3.phase >= 1 & !v3.d))
        ) : 0;
      1 : d;
    esac disabled_by {crash};
FAIRNESS running
```

Listing D.3: Caso 1: Commit Atómico (Parte 3 / 5)

```
MODULE RegularVoter(c, v0, v1, v2)
FAULT crash
  pre(1);
  effect(up = 0);
  restores(0);
VAR
  phase : {0, 1, 2};
  d : boolean;
  up : boolean;
INIT
  phase = 0 & up;
ASSIGN
  next(phase) :=
    case
      phase = 0 & c.up & c.phase = 1     : 1;
      phase = 0 & !c.up                  : 2;
      phase < c.phase & (c.phase = 2)    : 2;
      phase < v0.phase & (v0.phase = 2)  : 2;
      phase < v1.phase & (v1.phase = 2)  : 2;
      phase < v2.phase & (v2.phase = 2)  : 2;
      1                                  : phase;
    esac disabled_by {crash};
```

Listing D.4: Caso 1: Commit Atómico (Parte 4 / 5)

```
  next(d) :=
    case
      phase = 0 & c.up & c.phase = 1     : {0, 1};
      phase = 0 & !c.up                  : 0;
      phase < c.phase & (c.phase = 2)    : c.d;
      phase < v0.phase & (v0.phase = 2) : v0.d;
      phase < v1.phase & (v1.phase = 2) : v1.d;
      phase < v2.phase & (v2.phase = 2) : v2.d;
      1                                  : d;
    esac disabled_by {crash};
FAIRNESS running
```

Listing D.5: Caso 1: Commit Atómico (Parte 5 / 5)

```
MODULE main
VAR
  c : process Coordinator(rv0, rv1, rv2, rv3);
  rv0 : process RegularVoter(c, rv1, rv2, rv3);
  rv1 : process RegularVoter(c, rv0, rv2, rv3);
  rv2 : process RegularVoter(c, rv0, rv1, rv3);
  rv3 : process RegularVoter(c, rv0, rv1, rv2);

DEFINE S0 := ( (c.phase = 2 & c.d) ->
                    (rv0.phase != 0 & rv0.d)
                  & (rv1.phase != 0 & rv1.d)
                  & (rv2.phase != 0 & rv2.d)
             );

DEFINE S1 := ( (c.phase = 0 | c.phase = 1 | (c.phase = 2 & !c.d)) ->
                    (rv0.phase != 2 | !rv0.d)
                  & (rv1.phase != 2 | !rv1.d)
                  & (rv2.phase != 2 | !rv2.d)
             );

LTLSPEC G (S0 & S1)
```

Listing D.6: Caso 2: Elección de líder (Parte 1 / 3)

```
MODULE Host(host_id, pred, N, token, counter)
FAULT bizantina
  pre(1);
  effect(max = {0,1,2,3}, dist = {0,1,2,3}, counter = 0);
  restores(1);
DEFINE
  id := host_id;
VAR
  max  : 0 .. 3; -- big enough to hold K
  dist : 0 .. 3; -- big enough to hold N - 1
ASSIGN
  next(max) :=
    case
      (max < id) | (max != id & dist = 0) | (max = id & dist != 0) : id;
      (pred.dist + 1 < N) & (id < pred.max)                        : pred.max;
      ((pred.dist + 1 >= N) | (id > pred.id & id >= pred.max))     : id;
      1                                                            : max;
    esac;
  next(dist) :=
    case
      (max < id) | (max != id & dist = 0) | (max = id & dist != 0) : 0;
      (pred.dist + 1 < N) & (id < pred.max)                        : pred.dist + 1;
      ((pred.dist + 1 >= N) | (id > pred.id & id >= pred.max))     : 0;
      1                                                            : dist;
    esac;
```

Listing D.7: Caso 2: Elección de líder (Parte 2 / 3)

```
  next(token) :=
    case
      token = pred.id : id;
      1 : token;
    esac;

  next(counter) :=
    case
      token = pred.id & counter < 20 : counter + 1;
      1 : counter;
    esac;
FAIRNESS running

MODULE main
DEFINE
  N := 4; -- number of hosts
  K := 3; -- max id of any host
VAR
  token : 0 .. 3;
  counter : 0 .. 20;

  h0 : process Host(1, h3, N, token, counter);
  h1 : process Host(3, h0, N, token, counter);
  h2 : process Host(2, h1, N, token, counter);
  h3 : process Host(0, h2, N, token, counter);
```

Listing D.8: Caso 2: Elección de líder (Parte 3 / 3)

```
ASSIGN
  init(counter) := 0;

FAIRNESS running

DEFINE leader_elected := (h0.max = K & h1.max = K & h2.max = K & h3.max = K);
DEFINE consistent :=
  (h0.id = K -> h0.max = K & h0.dist = 0 & h1.max = K & h1.dist = 1
              & h2.max = K & h2.dist = 2 & h3.max = K & h3.dist = 3)
  & (h1.id = K -> h0.max = K & h0.dist = 3 & h1.max = K & h1.dist = 0
              & h2.max = K & h2.dist = 1 & h3.max = K & h3.dist = 2)
  & (h2.id = K -> h0.max = K & h0.dist = 2 & h1.max = K & h1.dist = 3
              & h2.max = K & h2.dist = 0 & h3.max = K & h3.dist = 1)
  & (h3.id = K -> h0.max = K & h0.dist = 1 & h1.max = K & h1.dist = 2
              & h2.max = K & h2.dist = 3 & h3.max = K & h3.dist = 0)
  ;

LTLSPEC G ((counter >= 10) -> (leader_elected & consistent))
```

Bibliografía

[1] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters 21, 181–185*, 1985.

[2] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Trans. Software Eng.*, 19(11):1015–1027, 1993.

[3] A. Arora and S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. *Proceedings of the 18th International Conference on Distributed Computing Systems*, 1998.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT press, 2008.

[5] J. Blanco, S. Smith, and D. Barsotti. *Cálculo de Programas*. Facultad de Matemática, Astronomía y Física, UNC, 2008.

[6] Marco Bozzano and Adolfo Villafiorita. The fsap/nusmv-sa safety analysis platform. *Int. J. Softw. Tools Technol. Transf.*, 9(1):5–24, 2007.

[7] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking*. Springer, 1999.

[8] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[9] IBM Corporation. Two-phase commit process. http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/index.jsp?topic=/com.ibm.db29.doc.admin/db2z_twophasecommit.htm.

[10] Oracle Corporation. Implementing two-phase commit. http://download.oracle.com/docs/html/A90849_01/ch7.htm.

[11] Edsger W. Dijkstra. Structured programming. circulated privately, aug 1969.

[12] A. Cimatti et al. Nusmv 2.4 user manual. http://nusmv.fbk.eu/NuSMV/userman/v24/nusmv.pdf.

[13] A. Cimatti et al. Integrating bdd-based and sat-based symbolic model checking, 2002.

[14] A. Cimatti et al. Nusmv 2: An opensource tool for symbolic model checking, 2002.

[15] Alliance for Telecommunications Industry Solutions. Telecommunications: Glossary of telecommunication terms. http://www.atis.org/glossary/.

[16] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Symposium on Principles of Programming Languages*, pages 111–122. ACM Press, 2004.

[17] Free Software Foundation. Lesser General Public License. http://www.gnu.org/licenses/lgpl-2.1.html.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. AddisonWesley Professional, 1994.

[19] Felix C. Gärtner. Specifications for fault-tolerance: A comedy of failures. Technical Report TUD-BS-1998-03, Darmstat University of Technology, Germany, 1998.

[20] Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, 1999.

[21] S. Ghosh and A. Gupta. An exercise in fault-containment: self-stabilizing leader election. *Information Processing Letters*, 59(5):281–288, 1996.

[22] Christoforos N. Hadjicostis. *Coding Approaches to Fault Tolerance in Combinational and Dynamic Systems*. The Springer International Series in Engineering and Computer Science, 2001.

[23] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, Computer Science Dept., 1994.

[24] Leslie Lamport and Michael Fischer. Byzantine generals and transaction commit protocols. Technical report, 1982.

[25] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.

[26] X. Lin and S. Ghosh. Maxima finding in a ring. In *Proc. of 28th Ann. Allerton Conf. on Computers, Communication, and Control*, pages 662–671, 1991.

[27] J.L. Lions et al. Ariane 501 inquiry board report, 1996. http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html.

[28] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. Wiley, 1999.

[29] Yukihiro Matsumoto. Ruby programming language. http://www.ruby-lang.org.

[30] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992.

[31] Takahiro Minamikawa, Tatsuhiro Tsuchiya, and Tohru Kikuno. Language and tool support for model checking of fault-tolerant distributed algorithms. In *PRDC '08: Proceedings of the 2008 14th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 40–47, Washington, DC, USA, 2008. IEEE Computer Society.

[32] K J Perry and S Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Trans. Softw. Eng.*, 12(3):477–482, 1986.

[33] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, 1977.

[34] Theo C. Ruys and Ed Brinksma. Model Checking: Verification or Debugging. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, pages 3009–3015.

[35] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, 1983.

[36] Nathan Sobo. Treetop. http://treetop.rubyforge.org/index.html.

[37] W. Steiner, J. Rushby, M. Sorea, and H. Pfeifer. Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. 2004.

[38] Inc. Sun Microsystems. Xa transactions. http://dev.mysql.com/doc/refman/5.0/en/xa.html.

[39] Wikipedia. Pipeline (software). http://en.wikipedia.org/wiki/Pipeline_(software).

[40] Tomoyuki Yokogawa, Tatsuhiro Tsuchiya, and Tsuchiya Kikuno. Automatic verification of fault tolerance using model checking. In *PRDC*, pages 95–102, 2001.