Universidad Nacional de Córdoba.
Facultad de Matemática, Astronomía y Física

Trabajo final de grado

Falluto2.0 Un Model Checker para la verificación automática de sistemas tolerantes a fallas.

Raúl Monti
Directores:
Dr. Pedro R. D'Argenio,
Dr. Nazareno Aguirre

Córdoba, 14 de febrero de 2013

Resumen

Los sistemas computacionales juegan roles determinantes en muchas áreas de nuestra vida cotidiana. En algunos casos la dependencia hacia estos sistemas es crítica, y el mal funcionamiento de los mismos puede acarrear grandes perdidas económicas o hasta de vidas humanas.

Las fallas en el diseño del sistema, como así también aquellas fallas causadas por el entorno de ejecución, pueden llevar a un comportamiento erroneo del mismo.

Los sistemas tolerantes a fallas son capaces de lidiar con la presencia de fallas y sobrellevar estas situaciones que en otros casos llevarían al mal funcionamiento de los mismos. Dar la capacidad de tolerar fallas a un sistema computacional no es sencillo y es muy propenso a introducir errores. La verificación de la corrección de estos sistemas es entonces de vital importancia.

El model checking es una técnica formal que se encarga de verificar exhaustivamente que el modelo de un sistema cumpla con un conjunto de propiedades.

En este trabajo se presenta un lenguaje para el modelado de sistemas tolerantes a fallas y la especificación de propiedades sobre los mismos. Se provee además la herramienta de verificación Falluto2.0, construida como front-end para el model checker simbólico NuSMV, la cual utiliza el lenguaje descripto. Se analiza la simplicidad y practicidad del lenguaje como así también su corrección. Por último se observa el uso de la herramienta sobre dos casos particulares de modelado de sistemas tolerantes a fallas.

ABSTRACT

AGRADECIMIENTOS

Índice general

Capítulo 1

Introducción

Es fácil notar la amplia dependibilidad que las personas hemos formado alrededor de dispositivos computacionales. A medida que crece la confianza hacia estos dispositivos para la realización de diferentes tareas, crece también el peligro que puede acarrear la ocurrencia de una falla en los mismos. En algunos casos, las actividades a las que son dedicados estos sistemas son actividades de bajo riesgo, como por ejemplo en un reloj de pulsera o un reproductor de música, y el incorrecto funcionamiento de los mismos no ocasiona daños mayores. En otros casos las actividades realizadas son de carácter crítico, como es por ejemplo en el caso de controladores de vuelo, o controladores de compuertas de contención fluvial. Es en estos últimos donde el incorrecto funcionamiento del sistema puede provocar grandes monetarias o hasta de vidas humanas.

Podemos considerar a la falla en una componente de hardware o software como una desviación de su función esperada. Las fallas pueden surgir durante todas las etapas de evolución del sistema computacional - especificación, diseño, desarrollo, elaboración, ensamblado, instalación- y durante toda su vida operacional [1] (debido a eventos externos). Este comportamiento fuera de lo normal puede llevar a un falla funcional del sistema, provocando que se comporte de manera incorrecta, o simplemente deje de funcionar.

Es importante entonces, para lograr una mayor confiabilidad del software (confiabilidad de que se comporte como su especificación describe) tomar acción sobre la ocurrencia de estas fallas. Existen diferentes enfoques para tratar con fallas. Uno de ellos es elaborar sistemas tolerantes a fallas. A diferencia de otros enfoques en los que se busca eliminar o disminuir la ocurrencia de fallas, en estos sistemas se busca disminuir los efectos de las fallas y en

el mejor de los casos recuperarse de estos y evitar que acarreen en fallas funcionales del sistema.

Queda claro entonces que un sistema tolerante a fallas provee grandes ventajas en comparación a uno que no contempla la ocurrencia de las mismas. Al igual que con el resto de los sistemas computacionales, es conveniente comprobar la corrección de los sistemas tolerantes a fallas. El diseño de algoritmos de tiempo real distribuidos tolerantes a fallas es notoriamente difícil y *propenso a errores*: la combinación de la ocurrencia de fallas, conviviendo con eventos concurrentes, y las variaciones en las duraciones de tiempos reales llevan a una explosión de estados que genera una gran demanda a la capacidad intelectual del diseñador humano [2].

En un mundo idealizado, los algoritmos son derivados por un proceso sistemático conducido por argumentos formales que aseguran su corrección respecto a la especificación original. 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. La exploración contra escenarios puede ser parcialmente automatizada usando un simulador y prototipos ágiles y esta automatización puede llegar a incrementar el número de escenarios que serán examinados y la confiabilidad de la examinación. La examinación automática de escenarios puede ser llevada a un nivel aún más avanzado usando *Model Checking* [2].

En ciencias de la computación *Model Checking* refiere al siguiente problema: dada una estructura formal del modelo de un sistema, y dada una propiedad escrita en alguna lógica específica, verificar de manera automática y exhaustiva si el sistema satisface la propiedad. El sistema normalmente representa a un componente de hardware o software, y la fórmula a cumplirse representa una propiedad de *safety* o *liveness* que se desea verificar que el sistema cumpla, y de esta manera incrementar la confiabilidad sobre el mismo.

El reducido nivel de interacción que presenta este método con el usuario es visto como un ventaja para la aplicación en la industria, ya que incrementa las posibilidad des ser usad por individuos no expertos [3]. Sin embargo es preciso modelar el sistema y definir las propiedades, lidiando mientras tanto con el principal problema del Model Checking: *la explosión de estados* debido al incremento exponencial de los mismos a raíz de la introducción de variables en la especificación del sistema.

Es objetivo de este trabajo elaborar una herramienta que contribuya a

disminuir el esfuerzo y a evitar la introducción de errores en el modelado y verificación mediante model checking de sistemas tolerantes a fallas. Buscamos con nuestra herramienta evitar tanto la introducción de errores en el modelado como así también contener la explosión de estados de la cual sufre el model checking. Queremos también que nuestro lenguaje de modelado ofrezca facilidades al momento de especificar las propiedades a verificar, de manera especial en el caso en que las mismas involucren la ocurrencia de fallas.

Para ello presentamos la herramienta de model checking *Falluto2.0*, orientada a la verificación de sistemas tolerantes a fallas. Esta herramienta presenta una capa de abstracción sobre NuSMV [4], un model checker basado en diagramas de decisión binaria. Falluto2.0 presenta un lenguaje de carácter declarativo para la introducción de fallas en el modelado del sistema, generando un marco de seguridad contra la introducción de errores evitando que el usuario deba explicitar el funcionamiento de la falla dentro del modelo. Este trabajo se presenta como extensión del trabajo realizado por Edgardo Hames [5], y toma como base los conocimientos adquiridos a partir del trabajo de características similares realizado por Nicolás Bordenabe [6].

Capítulo 2

Tolerancia a fallas

Mejorar la confiabilidad del sistema (el grado de confianza que se puede poner de manera justificada sobre el sistema) es usualmente presentado como el principal beneficio de la tolerancia a fallas. Introduciremos al comienzo de este capítulo los conceptos básicos de lo que entenderémos como *falla* en un sistema. Hablaremos luego de lo que significa que un sistema *tolere fallas*, y concluiremos planteando el contexto en el cual nuestro trabajo tratará la tolerancia a fallas.

2.1. Fallas

Intuitivamente, y dentro del contexto que nos compete, podemos definir a una *falla* ("fault" en inglés) en una componente de hardware o software como una perturbación de su estado la cual pone en peligro su correcto funcionamiento. Las fallas pueden ser introducidas en cualquier etapa de la evolución del sistema computacional en cuestión -especificación, diseño, desarrollo, elaboración, ensamblado, e instalación- y pueden ocurrir en cualquier momento durante su vida operacional [10].

Una definición ligeramente más formal nos sugiere definir el termino *falla* basado en la observación de que los sistemas cambian su estado como resultado de dos clases de eventos muy similares: operaciones normales de sistema y ocurrencia de fallas. Por lo tanto, un falla puede ser modelada como una no deseada (pero sin embargo posible) transición de estado en un proceso [9].

Denotaremos con el término *error* (en inglés "failure") al efecto no deseado que puede llegar a producir una falla en el comportamiento del sistema en

cuestión. Queda así distinguida la falla de su posible efecto negativo sobre el comportamiento del sistema.

2.2. Sistemas tolerantes a fallas

Existen sistemas diseñados para ser tolerantes a fallas: ellos o bien exhiben un comportamiento bien definido ante la ocurrencia de fallas, o bien enmascaran la falla de la componente al usuario, es decir continúan proveyendo su servicio estándar especificado a pesar de la ocurrencia de las fallas en la componente [11]. Podemos entonces decir de manera vaga que la tolerancia a fallas es la habilidad que posee un sistema de comportarse de una manera bien definida ante la ocurrencia de una falla. En el momento de diseñar la tolerancia a fallas, un primer prerrequisito es especificar la clase de falla que será tolerada. Tradicionalmente esto se lograba usando como base alguno de los modelos de fallas estándares (crash, fail-stop, etc...), sin embargo puede hacerse de manera más concisa especificando clases de fallas. El siguiente paso es enriquecer el sistema bajo consideración con componentes o conceptos que provean protección contra las fallas de una clase específica [10].

En este trabajo podremos distinguir dos clases de fallas en particular. Un primer grupo de fallas es de tipo *permanente*: éstas comúnmente representan fallas reales causadas por problemas irreversibles en la componente. Una vez que una falla permanente ocurre, permanece activa y afectando al sistema durante el resto de su ejecución. Por otro lado, un segundo grupo de fallas se caracteriza por ser de duración instantánea. Afectan el estado del sistema solo en los puntos específicos de se ejecución en los cuales estas fallas ocurren. Por lo tanto su efecto sobre el sistema solo resulta como consecuencia de los cambios particulares ocasionados en el momento de ocurrencia de la falla. Las mismas pueden repetirse indefinidamente durante la ejecución del sistema, permaneciendo activas solo en el momento en que afectan al sistema. Diremos que estas fallas son de clase *transitoria*.

Al establecer el grado de tolerancia de fallas de un sistema, no basta solo con indicar las características intrínsecas de las fallas mismas, sino que es de interés presentar el contexto de ocurrencia de estas. Por ello, al tratar con fallas, tomamos en cuenta diferentes condiciones y suposiciones sobre la ocurrencia de las mismas durante la ejecución del sistema. Muchas veces es de interés por ejemplo evitar tratar situaciones en las que las fallas ocurren constantemente, ya que esto suele presentar un escenario irreal. Otras veces

la ocurrencia de fallas en el modelo se observa en cantidades finitas y por lo tanto buscaremos poder establecer estos escenarios de análisis también. Todas estas condiciones nos permiten generar un contexto sobre el interleaving de las transiciones de falla y transiciones normales o buenas durante la ejecución del sistema. A partir de esto es que podemos analizar la tolerancia a fallas de manera mas específica y certera. La tolerancia a fallas de nuestros sistemas quedará entonces caracterizada no solo por el tipo de fallas sino que también a partir de las condiciones de ocurrencia de las mismas.

2.3. Discusión

Tomando como respaldo lo analizado en este capítulo, es que tomaremos una serie de decisiones sobre nuestro trabajo.

En este trabajo entonces no realizaremos mayor diferencia formal entre transiciones buenas del sistema y transiciones ocasionadas por fallas. Ambas serán consideradas como transiciones posibles del sistema que podrán afectar o no el estado del mismo. Esto difiere del punto de vista tomado por otros trabajos como [17, 18, etc...]

Distinguiremos además como dijimos dos clases de falla, las de tipo *Permanente* y las de tipo *Transitorio*, y la tolerancia de los sistemas modelados con respecto a ambas clases de fallas será estudiada bajo ciertas condiciones de ocurrencia e interleaving entre estas y acciones normales.

Es importante aclarar que si bien nos ocuparemos del modelado del sistema en general y nos centraremos por momentos en la inyección de fallas sobre el mismo, no nos detendremos en temas específicos al diseño de la tolerancia a fallas.

Capítulo 3

Model Checking

En el diseño de software y hardware de sistemas complejos, se consume más tiempo y esfuerzo en verificación que en construcción. Se entiende que la aplicación de técnicas reduce y aligera los esfuerzos en verificación a la vez que acrecientan su cobertura. Los métodos formales ofrecen un gran potencial para obtener una integración temprana de la verificación en el proceso de diseño, y para proveer de técnicas de verificación más afectivas y reducir el tiempo invertido en aplicarlas [12].

El *Model Checking* es una técnica formal que permite la verificación exhaustiva de modelos finitos de sistemas. Logra esta verificación explorando todos los estados posibles del sistema modelado con la intención de constatar la veracidad un conjunto de propiedades sobre el mismo.

En este capítulo comenzaremos revisando algunos conceptos básicos para entender el funcionamiento de los *model checkers* (programas utilizados para la verificación mediante model checking), y concluiremos presentando el proceso de model checking junto con algunas características de este método.

3.1. Sistemas de transiciones

Labelled transition systems - LTS

Llamamos *sistema de transiciones etiquetadas* (en inglés "labeled transition systems") a un tipo de maquina abstracta usada entre otras cosas para el modelado de sistemas computacionales concurrentes. Este sistema de representación está compuesto por un conjunto de estados, un conjunto

de etiquetas o nombres, y una relación ternaria explicando las transiciones etiquetadas desde un estado hacia otro del sistema.

Formalmente podemos decir que un LTS es una tres-upla $M = (S, S_0, L, R)$ donde:

- S es un conjunto de estados
- $S_0 \subseteq S$ es un conjunto de estados iniciales
- L es un conjunto de etiquetas (nombres de transiciones)
- $R \subseteq S \times L \times S$ es una relación ternaria de transiciones etiquetadas

Notemos entonces que si $s1$ y $s2$ son elementos en S, l es un nombre en L, y $(s1, l, s2) \in R$, entonces estamos indicando que existe una transición con nombre l desde el estado $s1$ al estado $s2$. Usualmente, las etiquetas se utilizan para nombrar la acción que la transición realiza. De esta manera podemos modelar sistemas computacionales tomando cada elemento en S como un estado particular del sistema y definiendo relaciones etiquetadas entre los mismos para representar su comportamiento.

En particular nos interesa para este trabajo la capacidad que otorga este formalismo para representar sincronización entre acciones de distintas componentes del sistema. En el ejemplo de la figura 3.1 encontramos dos componentes, un productor y un consumidor, ambas con su representación en términos de LTS. Podemos, a partir de ellas, definir un sistema concurrente sincronizado en el cual las transiciones de igual nombre en cada componente deben ser ejecutadas de manera sincronizada, mientras que las transiciones propias de cada componente pueden ejecutarse independientemente. Vemos el sistema resultante en la figura 3.2. En ella la transición punteada y etiquetada con 'Listo' representa la acción sincronizada entre el productor y el consumidor. A su vez podemos ver en el estado punteado un caso de *interleaving* entre los procesos sincronizados. Allí podemos elegir entre darle paso a la acción 'Producir' del productor, o dejar que el consumidor realice la acción 'Consumir'.

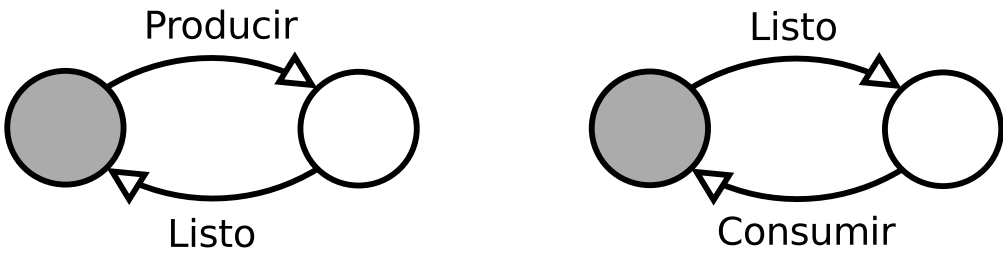

Figura 3.1: El productor (izquierda) y el consumidor (derecha).

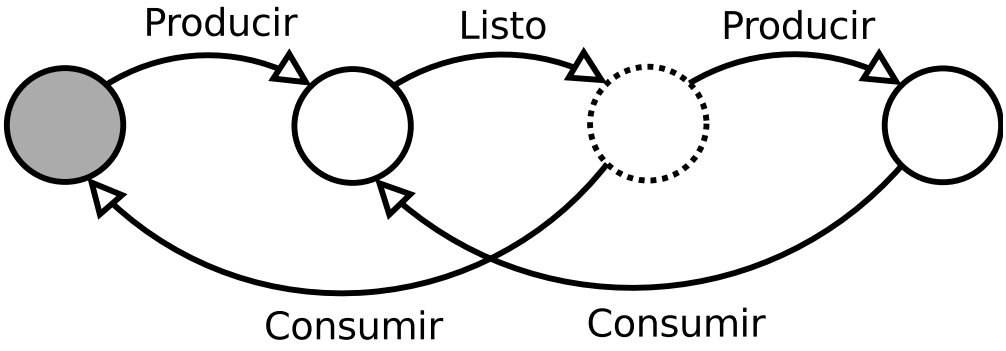

Figura 3.2: Productor y consumidor de la figura 3.1 sincronizados.

Estructuras de Kripke

Una forma alternativa de describir comportamiento es a través de estructuras de Kripke. Las estructuras de Kripke son otro variante de la máquina de estados que en lugar de centrarse en la actividad observable, como hacen los LTS, se enfoca en las propiedades que satisface cada estado. Una estructura de Kripke consiste en un conjunto de estados, un conjunto de transiciones entre esos estados, y una función que etiqueta cada estado con un conjunto de propiedades que son verdaderas en este estado. Los caminos en estas estructuras modelan la ejecución del sistema [13]. Estas estructuras son lo suficientemente expresivas como para captar aspectos de lógicas temporales tales como LTL y CTL que son lógicas mayormente usadas por los model checkers para expresar las propiedades esperadas del sistema.

Formalmente podemos definir una estructura de Kripke como sigue [13]:

Sea AP un conjunto de proposiciones atómicas. Una estructura de Kripke M sobre AP es una cuatro-upla (S, S_0, R, L) tal que:

1. S es un conjunto finito de estados.
2. $S_0 \subseteq S$ es un conjunto de estados iniciales.
3. $R \subseteq S \times S$ es una relación de transición que debe ser total, es decir, para cada estado $s \in S$ hay un estado $s' \in S$ tal que $(s, s') \in R$.
4. $L : S \to 2^{AP}$ es una función que etiqueta cada estado con el conjunto de proposiciones atómicas verdaderas en ese estado.

Una ejecución del sistema desde un estado s es representado en la estructura M como una secuencia infinita $\pi = s_0 s_1 s_2 ...$ tal que $s_0 = s$ y $(s_i, s_{i+1}) \in R$ para todo $i \geq 0$.

Notemos que podemos traducir la representación en términos de LTS de un sistema a una representación en términoes de estructuras de Kripke equivalente de la siguiente manera. Sea $M_1 = (S_1, S_{1_0} R_1, L_1)$ un sistema descripto en LTS, entonces construimos la estructura de Kripke $M_2 = (S_2, S_{2_0}, R_2, L_2)$ sobre AP como sigue:

- $AP = \{action = e \mid e \in L_1 \cup \{null\}\}$
- $S_2 = S_1 \times (L_1 \cup \{null\})$
- $S_{2_0} = S_{1_0} \times \{null\}$

- $R_2 = \{(s, a) \to (s', b) \mid s \xrightarrow{b} s' \in R_1, a \in L \cup \{null\}\}$
- $L_2(s, a) = (action = a)$, para todo $(s, a) \in S_2$

Lo que hicimos fue entonces construir por cada estado s_i y etiqueta e en el LTS un estado en la estructura de Kripke que represente llegar al estado s_i usando la etiqueta e. Dado que en el inicio de las ejecuciones no realizamos acción alguna para llegar al estado inicial, es que hemos además definido para cada estado $s \in S_{1_0}$ un estado $(s, null)$ que lo represente en S_2. La función de relación se forma de manera intuitiva. El etiquetado indica qué acción se llevó a cabo para llegar a cada estado. Esto último, junto con el nombre del estado, dejan en claro la transición llevada a cabo en la ejecución del sistema definido en el LTS original. Vemos un ejemplo de traducción LTS-Kripke en la figura 3.3. Notemos que el sistema traducido puede ser depurado quitando estados no alcanzables como se muestra en la figura 3.3.C.

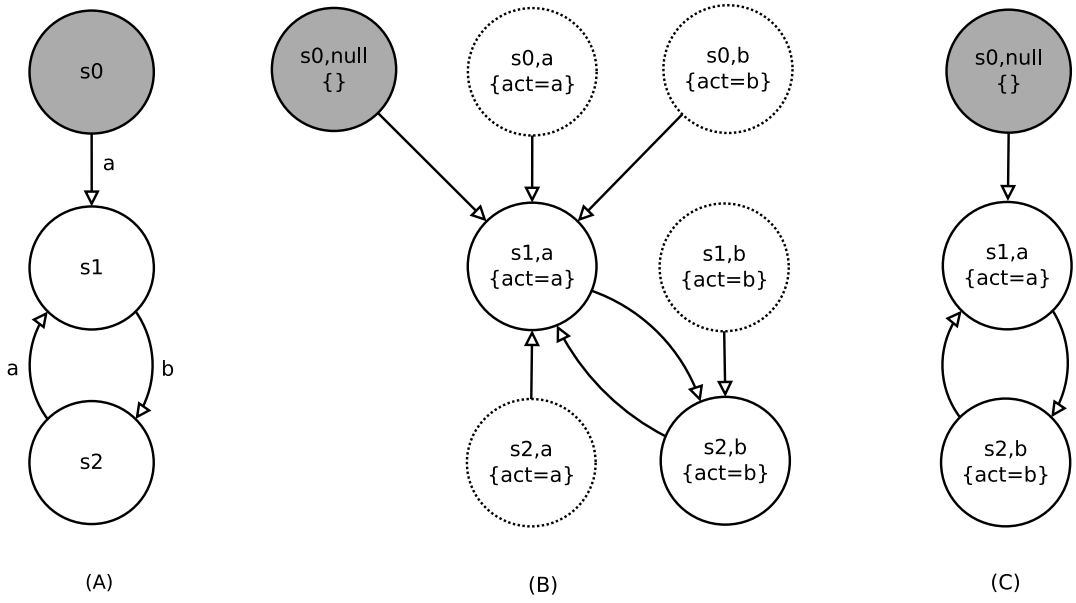

Figura 3.3: Ejemplo de traducción de un sistema LTS a estructuras de Kripke. (A) El LTS original; (B) su traducción a estructuras de Kripke; y (C) el fragmento alcanzable de dicha estructura de Kripke

3.2. De estructuras de Kripke a fórmulas lógicas

Si bien las estructuras de Kripke sirven para captar intuitivamente el comportamiento de los sistemas a modelar, los model checkers trabajan sobre el modelado del sistema en base a la lógica proposicional. A continuación veremos como lograr la interpretación de estructuras de Kripke usando fórmulas proposicionales. En lo que a nosotros concierne, la lógica proposicional estará comprendida por los conectivos lógicos usuales - $\neg, \wedge, \vee, \rightarrow, etc...$ -, y haremos uso también de los cuantificadores $\forall$ y $\exists$ sobre dominios finitos como generalización del $\wedge$ y del $\vee$ respectivamente.

Supongamos que queremos modelar un sistema P. Consideremos el conjunto de variables $V = \{v_0, v_1, ..., v_n\}$. Consideremos para el caso que todas estas variables toman valores de un domino finito D. Tenemos que una valuación de V es una función total sobre el dominio, la cual asigna a cada variable en V un valor en D. Notemos que dado que los valores de las variables del sistema son las que definen el estado del mismo en su totalidad, cada valuación estaría definiendo el estado del sistema. Por lo tanto un sistema P con variables V sobre el dominio D se puede representar con una estructura de Kripke $M = (S, S_0, R, L)$ sobre AP donde:

- $AP = \{v = d \mid v \in V, d \in D\}$.
- Cada estado $s \in S$ es una valuación sobre V.
- R explica la relación entre estas valuaciones, vale decir explica la transición entre los cambios de valores en las variables del sistema.
- $L(s)$ es el subconjunto de proposiciones en AP que son validas dada la valuación del estado s.

Dada un estado en la estructura de Kripke, es decir una valuación $s : V \rightarrow D$, podemos escribir una fórmula sobre las variables en V tal que sea válida solo para esta valuación [13]. La fórmula sería:

$$(v_0 = s(v_0))\&(v_1 = s(v_1))\&...\&(v_n = s(v_n))$$

Dado que en general una fórmula puede ser satisfecha por diferentes valuaciones, podemos a partir de ella definir el conjunto de estados que la satisfacen. Así por ejemplo podemos definir una fórmula cuyos estados que la

satisfacen sean solo los estados iniciales del sistema (S_0 en la estructura de Kripke que lo modela).

Además de definir los estados, necesitamos poder especificar también transiciones como fórmulas interpretadas de primer orden. Para ello debemos lograr a partir de una fórmula representar la relación entre una valuación actual y la siguiente (estado actual y estado sucesor). Necesitaremos entonces otro conjunto de variables V' el cual representen las variables en estado sucesor. De esta forma, dada la fórmula de transición F_r sobre el conjunto de variables $V \cup V'$ diremos que s' es estado sucesor de s, es decir que $(s, s') \in R$, si y solo si F_r se satisface al evaluar todas sus variables en V según s y todas sus variables en V' según s'.

Por último, las proposiciones en AP nos permiten definir propiedades sobre los estados. Recordemos que las proposiciones atómicas en AP son de la forma $v = d$ con $v \in V$ y $d \in D$. Entonces tenemos que una proposición $v = d$ es válida en el estado s si y solo si $d = s(v)$. En tal caso tenemos que $v = d \in L(s)$.

3.3. Representación de funciones booleanas en BDD

Las fórmulas proposicionales (o funciones booleanas) son un fuerte formalismo para representar los sistemas de transiciones y razonar acerca de propiedades sobre ellos. Se busca entonces poseer una representación eficiente de estas funciones para poder abarcar sistemas considerablemente complejos. NuSMV, como así también muchos otros model checkers, logra esta representación a partir de *diagramas de decisión binaria (BDD)*. Los BDD presentan muchas ventajas en cuanto a eficiencia en el cálculo y en el espacio de almacenamiento con respecto a otras representaciones como pueden ser las tablas de verdad o subclases de la fórmula proposicional como la forma normal conjuntiva.

Una función booleana de n argumentos es una función $f : \{0,1\}^n \to \{0,1\}$. Podemos definir a los BDD como un subconjunto de grafos dirigidos acíclicos finitos con las siguientes características [15]:

- Poseen un único nodo inicial (un único nodo al cual no llega ninguna arista).

- Todas sus hojas son etiquetadas ya sea con '0' o con '1'.
- Todos sus nodos que no son hojas están etiquetados con el nombre de una variable booleana.
- Cada nodo (excepto las hojas) posee dos aristas salientes hacia otros nodos del diagrama, una etiquetada con '0' y otra etiquetada con '1'.

Un BDD B define en sus hojas una única función booleana sobre las variables en sus nodos de la siguiente manera: arrancando por el nodo inicial y dada una valuación $V = (v_1, v_2, ..., v_n)$ sobre el conjunto de variables en los nodos, si v_i es 0 entonces tomo la arista saliente etiquetada con '0', de lo contrario tomo la etiquetada con '1'. Este paso se repite en cada nodo hasta llegar a un hoja. El valor de la hoja es el valor, para esa valuación, de la función booleana representada.

Podemos reducir el tamaño de los BDD logrando una mejora de eficiencia en el espacio de memoria necesario para la representación de las funciones booleanas. Para ello usamos los siguientes tres métodos [15]:

R1– *Remoción de terminales duplicados.* Si el BDD posee más de un nodo terminal '0', entonces redirigimos todas las aristas apuntando a esos nodos a uno solo de ellos y eliminamos el resto. Hacemos lo mismo con los nodos terminales '1'.

R2– *Remoción de verificaciones innecesarias.* Si ambas aristas salientes de un nodo n apuntan a un mismo nodo m entonces eliminamos el nodo n redirigiendo todas sus arista entrantes al nodo m.

R3– *Remoción de no terminales duplicados.* Si dos nodos no terminales n y m son raíces de sub-BDDs estructuralmente idénticos, entonces eliminamos uno de ellos y redirigimos todas sus aristas entrantes al otro.

Estas reducciones no afectan la representación del BDD con respecto a la fórmula booleana original, y presenta una gran ventaja con respecto al espacio necesario para almacenar tablas de verdad por ejemplo. En la figura 3.4 podemos ver un ejemplo de aplicación de estas reducciones. En la figura (A) vemos el BDD original. La figura (B) es el resultado de unificar las hojas según R1. La figura (C) es el resultado de eliminar uno de los nodos de etiqueta 'y' según R3. Finalmente (D) es el resultado de aplicar la regla R2 eliminando el nodo de etiqueta 'x'.

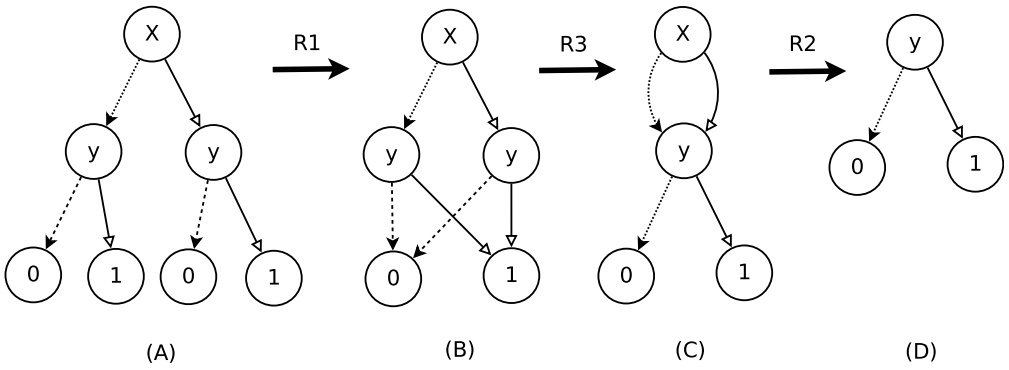

Figura 3.4: Reducción de un BDD según reglas R1,R2 y R3

Si bien con estas reglas se logra reducir en gran proporción cierto conjunto de BDDs, otros BDDs no son tan susceptibles a ser reducidos de esta manera. Un efecto negativo visible en mucho de estos es la reutilización en sus nodos de una misma variable booleana, lo cual implica revisar el valor de aquellas más de una vez. Otro defecto surge de que diferentes BDDs pueden representar una misma función booleana, por lo cual comparar la equidad de las funciones booleanas representadas en BDDs requiere de mucho trabajo.

Todas estas complicaciones logran ser evitadas usando un subconjunto de BDDs llamados *diagramas de decisión binaria ordenados* (OBDD). Los OBDD imponen un orden sobre la aparición de las variables en sus nodos, evitando de este modo los BDDs con más de una aparición de la misma variables. Podemos dar la siguiente definición de un OBDD [15]:

Sea $[x_1, x_2, ..., x_n]$ *una lista ordenada de variables sin duplicados y sea* B *un BDD cuyas variables pertenecen a la lista. Decimos que* B *tiene el ordenamiento* $[x_1, x_2, ..., x_n]$ *si todas las variables en los nodos de* B *ocurren en la lista, y para cada ocurrencia de* x_i *seguida de* x_j *en cualquier camino sobre* B, $i < j$. *Decimos entonces que un OBDD es un BDD que posee un orden para alguna lista de variables.*

Notemos que como corolario de la definición obtenemos que una variable no puede ocurrir más de una vez en el OBDD. Estos diagramas poseen además la cualidad de que para cada función booleana y orden de variables existe un único OBDD reducido que la representa. Osea que si B_1 y B_2 son

dos OBDD reducidos con orden de variables compatible que representan a la función booleana f entonces B_1 y B_2 son idénticos. Esto lleva a que la comparación por igualdad entre los mismos se reduzca a una simple comparación de isomorfismo. Cuando no podemos seguir reduciendo un OBDD según las reglas R1, R2 y R3, decimos que está en su forma canónica. Son de interés especial las formas canónicas de la figura 3.5 debido a que nos permiten identificar las siguientes situaciones:

- *Validez de una función:* podemos verificar la validez de una función booleana (verificar si la función siempre computa 1) de la siguiente manera: reducimos su OBDD a su forma canónica, si el resultado de la reducción es B_1 entonces la función es válida.

- *Implicación:* podemos verificar si la función $f(x_1, x_2, ..., x_n)$ implica la función $g(x_1, x_2, ..., x_n)$ (es decir si f computa 1 entonces g también) reduciendo el OBDD obtenido de $f \wedge \neg g$. Si el OBDD canónico es B_0 entonces la implicación es verdadera.

- *Satisfactibilidad:* podemos verificar satisfactibilidad de una función $f(x_1, x_2, ..., x_n)$ (es decir si f computa 1 para alguna asignación de ceros y unos a sus variables) reduciendo su OBDD a forma canónica y verificando que ésta no sea B_0

El orden de variables elegido toma vital importancia, ya que de él depende el tamaño del OBDD. Existen algoritmos que logran un ordenamiento inteligente de las variables para mantener acotado el tamaño de almacenamiento necesario para el OBDD. Así también existen algoritmos que hacen eficiente la reducción de OBDDs a su OBDD canónico, como también algoritmos que permiten computar conjunción, disyunción y negación de OBDDs con considerable eficiencia.

El uso de BDDs produjo un gran salto en el model checking llevando en principios de los noventa a lograr verificar sistemas con espacios de estados mucho más grandes que con métodos anteriores. El model checking usando OBDDs es llamado *Model Checking Simbólico* y debe su nombre a que sugiere no representar cada estado por separado, sino que los OBDDs representan los conjuntos de estados de manera simbólica. Veremos ahora como representar estos conjuntos de estados en OBDDs y como representar la relación de transición entre ellos.

Figura 3.5: OBDDs canónicos de relevancia

Recordemos que en nuestro modelo de sistema como estructura de Kripke habíamos definido un conjunto de proposiciones atómicas sobre las variables del sistema, que representábamos a cada estado como una valuación sobre las variables del sistema, y que la función de etiquetado L nos proporcionaba el subconjunto de AP que era válido en cada estado. Supondremos que hay un orden sobre las proposiciones en AP dado como $(x_1, x_2, ..., x_n)$ donde cada x_i representa una proposición atómica y representaremos cada estado s como un vector $(v_1, v_2, ..., v_3)$ donde $v_i \in \{0, 1\}$ y $v_i = 1$ si y solo si $x_i \in L(s)$. Vemos entonces que los estados son representados por la función característica inducida por L. Es decir, la función booleana que representa un estado es aquella que valuada en $(v_1, v_2, ..., v_3)$ computa el valor 1, y para cualquier otro valuación de sus variables computa el valor 0. En términos de OBDDs, el OBDD que representa al estado s es aquel que codifica la función booleana :

$$l_1 \wedge l_2 \wedge ... \wedge l_n$$

donde l_i es x_i si $x_i \in L(s)$ y $\neg x_i$ de lo contrario. Luego para un conjunto de estados $S = s_1, s_2, ..., s_m$ la representación en OBDD esta dada por aquel que codifica a la función booleana

$$(l_{11} \wedge l_{12} \wedge ... \wedge l_{1n}) \vee (l_{21} \wedge l_{22} \wedge ... \wedge l_{2n}) \vee ... \vee (l_{m1} \wedge l_{m2} \wedge ... \wedge l_{mn})$$

donde $(l_{i1} \wedge l_{i2} \wedge ... \wedge l_{in})$ representa al estado s_i.

Para el caso de la relación de transición recordemos que ésta es un subconjunto sobre $S \times S$. De nuevo podemos inducir la función booleana representante a partir de la función de etiquetado L. De este modo una relación $s \rightarrow s'$ en R es representada por los vectores $(v_1, v_2, ..., v_n), (v'_1, v'_2, ..., v'_n)$, donde $v_i = 1$ si $p_i \in L(s)$ y 0 en caso contrario, y $v'_i = 1$ si $p_i \in L(s')$ y

0 en caso contrario. Una transición queda representada por el OBDD de la función booleana

$$(l_1, l_2, ..., l_n) \wedge (l'_1, l'_2, ..., l'_n)$$

y el conjunto representando el total de la relación R se logra a partir de la conjunción de cada una de estas fórmulas.

3.4. Lógicas temporales

Las lógicas temporales son lógicas modales que permiten describir y razonar sobre proposiciones en términos del tiempo. En el caso del model checking, son de gran utilidad para la especificación de las propiedades a verificar sobre el modelo del sistema. A continuación presentaremos dos lógicas temporales de interés para este trabajo. Ambas lógicas difieren en expresividad, por lo que hacer uso de las dos nos permite una mayor versatilidad al momento de especificar las propiedades deseadas.

LTL

La *Lógica de Tiempo Lineal (LTL)* permite razonar sobre ejecuciones lineales a través del tiempo. Estas ejecuciones solo permiten ver los cambios de estado que se producen a lo largo del tiempo y cómo estos se ordenan temporalmente, pero no permiten ver el transcurso preciso del tiempo. Así, cada momento vvisible representa una configuración en el estado del sistema y cada salto en el tiempo representa una transición desde un estado del sistema a uno nuevo. Estas lógicas están compuestas por un conjunto finito de proposiciones atómicas AP, los conectivos booleanos $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ y los conectivos temporales G, F, X, U, R. Estos últimos conectivos se corresponden con las palabras en idioma inglés Globally, Finally, ne$X$t, Until y Release. A continuación damos una descripción intuitiva del significado de cada conectivo:

- $\mathsf{G}\,\phi$ expresa que en todo momento durante la ejecución se satisface la fórmula ϕ.
- $\mathsf{F}\,\phi$ expresa que en algún momento en el futuro se satisface la fórmula ϕ.

- $\mathsf{X}\,\phi$ expresa que ϕ se satisface en el momento inmediatamente posterior al actual.
- $\phi\,\mathsf{U}\,\psi$ expresa que ψ se satisface en algún momento, y para todo momento anterior a aquel ϕ se satisface.
- $\phi\mathsf{R}\psi$ expresa que o bien ϕ no se satisface nunca y ψ se satisface siempre, o bien ψ se satisface en cada momento hasta que ϕ se satisfaga.

Usando LTL podemos expresar propiedades de *liveness* y *safety* de nuestro sistema de manera sencilla. Por ejemplo para expresar *"En algún momento algo bueno sucederá"* usamos F "algo bueno", o para expresar *"En ningún momento algo malo sucede"* usaríamos $G\,\neg$ "algo malo".

Las fórmulas LTL se interpretan usualmente sobre ejecuciones de estructuras de Kripke, y si queremos saber si la fórmula ϕ se satisface en un sistema representado por la estructura de Kripke M, basta con ver que el lenguaje de M (i.e. todas las ejecuciones posibles en M) satisfaga ϕ. Definimos la semántica de fórmulas LTL como sigue:

Sea la palabra $\omega = s_1 s_2 s_3 ...$ de valuaciones en AP. Definimos la relación de satisfactibilidad $\models$ de una fórmula LTL con respecto a la palabra ω a partir de las siguientes reglas:

- $\omega \models p$ si $p \in L(\omega[0])$
- $\omega \models \neg p$ si no $\omega \models p$
- $\omega \models \phi \vee \psi$ si $\omega \models \phi$ o $\omega \models \psi$
- $\omega \models \mathsf{X}\,\phi$ si $\omega[1...] \models \phi$
- $\omega \models \phi\,\mathsf{U}\,\psi$ si existe $i \geq 0$ tal que $\omega[i...] \models \psi$ y para todo $0 \leq k < i$, $\omega[k...] \models \phi$

donde $\omega[i...]$ es el i-ésimo sufijo de ω.

Notar que los demás conectivos son derivados de aquellos para los que hemos definido las reglas de satisfactibilidad: $F\phi \equiv trueU\phi, G\phi \equiv \neg F\neg\phi$ y $\phi R\psi \equiv \neg(\neg\phi U\neg\psi)$.

Muchas veces es de interés en model checking definir propiedades bajo *condiciones de equidad (en inglés "Fairness Conditions")* . Por ejemplo en el contexto del modelado de un planificador de tareas podemos requerir que el mismo atienda equitativamente a los diferentes procesos. LTL, a diferencia de CTL, nos da la posibilidad de definir estas equidades:

1. Equidad incondicional:

   $\mathsf{G}\ \mathsf{F}\ p$ (a menudo se cumple p)

2. Equidad fuerte:

   $\mathsf{G}\ \mathsf{F}\ q \rightarrow \mathsf{G}\ \mathsf{F}\ p$ (si q se satisface a menudo, entonces p también)

3. Equidad débil:

   $\mathsf{F}\ \mathsf{G}\ q \rightarrow \mathsf{G}\ \mathsf{F}\ p$ (si en algún momento q se satisface como invariante, entonces p debe satisfacerse a menudo)

En 2 y en 3, q podría verse como la condición de habilitación de p.

CTL

A diferencia de LTL, la *Lógica de árbol computacional (en inglés "Computation tree logic" (CTL))* considero toda posible bifurcación de la ejecución. Por lo tanto, una fórmula CTL puede hacer referencia a cualquier ejecución que se desprenda de cualquier estado considerado. Además de los conectivos lógicos y temporales introducidos en LTL, CTL implementa el uso de los cuantificadores A (para todo camino) y E (existe un camino). De hecho, CTL requiere que todo operador temporal esté cuantificado. Esto hace que las fórmulas sean interpretadas en estados en lugar de caminos y las cuantificaciones hacen referencia a la existencia o universalidad de los caminos partiendo de un estado. Así es que podemos especificar propiedades como $\mathsf{EX}\ p$ y $\mathsf{AG}\ \mathsf{EF}\ p$, las cuales no pueden ser especificadas en LTL ya que en ella no podemos hablar de la existencia de al menos un camino en el futuro en el cual se cumple p.

Los conectivos $\neg, \wedge, \mathsf{AX}, \mathsf{EX}, \mathsf{EU}$ comprenden un conjunto completo de conectivos para la lógica CTL dado que los demás pueden ser derivados a partir de ellos. Podemos decir que el siguiente es el significado intuitivo de estos conectivos:

- $\neg$ es la negación booleana usual.
- $\wedge$ es la conjunción booleana usual.
- $\mathsf{AX}\ \phi$ se cumple en un estado s si para cualquier ejecución, ϕ se satisface en todo estado sucesor de s.

- $\mathsf{EX}\ \phi$ se cumple en un estado s si existe al menos una ejecución con un estado sucesor a s y donde ϕ se satisface.
- $\mathsf{E}[\phi\ \ \mathsf{U}\ \ \psi]$ se cumple en un estado s si existe al menos una ejecución $s_1 s_2 ... s_n ...$ con $s_1 = s$ con un estado s_k donde ψ se hace verdadero y ϕ se satisface en cada estado $s_i, 1 \leq i \leq n-1$.

A continuación, utilizando estructuras de Kripke, definimos la semántica formal de CTL por inducción estructural sobre una fórmula ϕ. Sea la estructura de Kripke $M = (S, S_0, R, L)$ sobre AP, sea ϕ una fórmula CTL bien formada sobre AP, y sean $s \in S$ y $p \in AP$:

- $M, s \models p \quad \leftrightarrow \quad p \in L(s)$
- $M, s \models \neg\phi \quad \leftrightarrow \quad M, s \not\models \phi$
- $M, s \models \phi_0 \wedge \phi_1 \quad \leftrightarrow \quad M, s \models \phi_0$ y $M, s \models \phi_1$
- $M, s \models \mathsf{AX}\,\phi \quad \leftrightarrow \quad \forall (s, s') \in R,\ M, s' \models \phi$
- $M, s \models \mathsf{EX}\,\phi \quad \leftrightarrow \quad \exists (s, s') \in R$, tal que $M, s' \models \phi$
- $M, s \models \mathsf{E}(\phi_0\ \mathsf{U}\ \phi_1) \quad \leftrightarrow \quad \exists$ una ejecución ω definida por R tal que $\omega[0] = s$ y $\exists n \geq 0\ M, \omega[n] \models \phi_1$, y $\forall 0 < i < n\ M, \omega[i] \models \phi_0$

3.5. Características del Model Checking

Existen diferentes métodos para la verificación de sistemas complejos, entre ellos podemos destacar como principales la simulación, el testing, la verificación deductiva, y el model checking [13]. Tanto la simulación como el testing comprenden realizar experimentos antes de desplegar el sistema en el campo. Mientras que en el caso de la *simulación* se trabaja sobre una abstracción o modelo del sistema, en el *testing* se trabaja sobre el producto real. En cuanto a costo-eficiencia, estos métodos pueden ser ventajosos para detectar gran cantidad de errores. Sin embargo, revisar todas las posibles interacciones y potenciales errores usando simulación y testing es prácticamente imposible.

El término *verificación deductiva* normalmente refiere al uso de axiomas y reglas para probar la corrección del sistema. Este método si bien posee la ventaja de poder probar corrección sobre sistemas de estados infinitos no es muy utilizado fuera de casos críticos. Esto se debe a que requiere de gran cantidad de tiempo y de la conducción por parte de expertos en el campo del razonamiento lógico.

El *Model checking* es un método automático para la verificación de propiedades sobres sistemas concurrentes finitos. Trabaja sobre un modelo del sistema y explora exhaustivamente todos sus posibles estados con el fin de verificar si una propiedad especificada sobre el mismo es verdadera o no. Presenta ciertas ventajas sobre los métodos anteriormente mencionados:

- Detecta errores en etapas tempranas de diseño, evitando tener que replantear todo al encontrar estos errores en etapas posteriores.
- Gran parte de su proceso es automático, por lo cual no requiere de personal experto en campos de la matemática para llevar a cabo las tareas de verificación.
- Es exhaustivo con respecto al conjunto de estados del sistema.
- Ofrece clara evidencia del problema en el caso de encontrar que la propiedad deseada sobre el sistema no se cumpla.

Sin embargo el *model checking* sufre del problema de la explosión de estados. Esto implica que fácilmente se llegue a sistemas en los que la cantidad de estados es tan grande que supera los límites de memoria física del hardware sobre el que corre el programa de model checking. Muchos algoritmos y optimizaciones sobre los model checkers han ayudado a combatir este efecto, pero sin embargo el mismo persiste. Otra solución a este problema es llevar el modelado a una mayor abstracción con el fin de disminuir la cantidad de estados finales. Al hacer esto debemos tomar en cuenta que la abstracción debe asegurar que las propiedades que no se satisfacen en el sistema concreto, tampoco se satisfagan en el modelo abstracto.

Christel Baier y Joost-Pieter Katoen [14] identifican las siguientes tareas como pasos para realizar el *proceso de model checking* sobre el diseño de un sistema (ver también la figura 3.6):

1. Fase de modelado:
    - Modelar el sistema en consideración usando el lenguaje del model checker que se tenga a mano.
    - Realizar algunas simulaciones sobre el modelo como primera revisión y rápida aceptación del mismo.

- Describir las propiedades a verificar sobre el modelo usando el lenguaje específico para esta tarea.

2. Fase de ejecución:

   Ejecutar el model cheker para verificar la validez de una propiedad sobre el sistema modelado.

3. Fase de análisis:

   - Si la propiedad resultó ser válida $\longrightarrow$ en el caso de haber más propiedades a verificar, proceder con la verificación de las mismas.
   - Si la pripiedad fué refutada $\longrightarrow$
     a) analizar, a partir de simulación, el contraejemplo generado.
     b) refinar el modelo, diseño o propiedad.
     c) repetir todo el proceso.
   - La computadora se quedó sin memoria $\longrightarrow$ intentar reducir el modelo y empezar de nuevo.

En la fase de modelado podemos distinguir entonces por un lado el modelado del sistema y por otro la especificación de las propiedades. El modelado del sistema, si bien algunas veces suele ser una simple compilación al lenguaje de modelado específico de la herramienta de model checking, otras veces puede requerir depuración del modelo para eliminar características innecesarias que puedan ocasionar una explosión de estados a tal grado de agotarse la memoria. Por otro lado podemos destacar como asunto importante en la etapa de especificación de propiedades el hecho de lograr completitud sobre las características deseadas del sistema. Es decir que es esencial en esta etapa lograr expresar en las propiedades el conjunto completo de características que se desea que el sistema posea.

La fase de ejecución usualmente es automática y solo consta de proveer al model checker con el modelado del sistema y la especificación de una propiedad para que el mismo decida sobre su validez.

Por último, es en la fase de análisis donde podemos decidir ya sobre el resultado de la verificación. Se pueden presentar varios escenarios: por un lado puede que el proceso de verificación no se haya completado debido al agotamiento de memoria, en cuyo caso se deberá recurrir al refinamiento del sistema para conseguir disminuir la explosión de estadosPuede ser el caso de

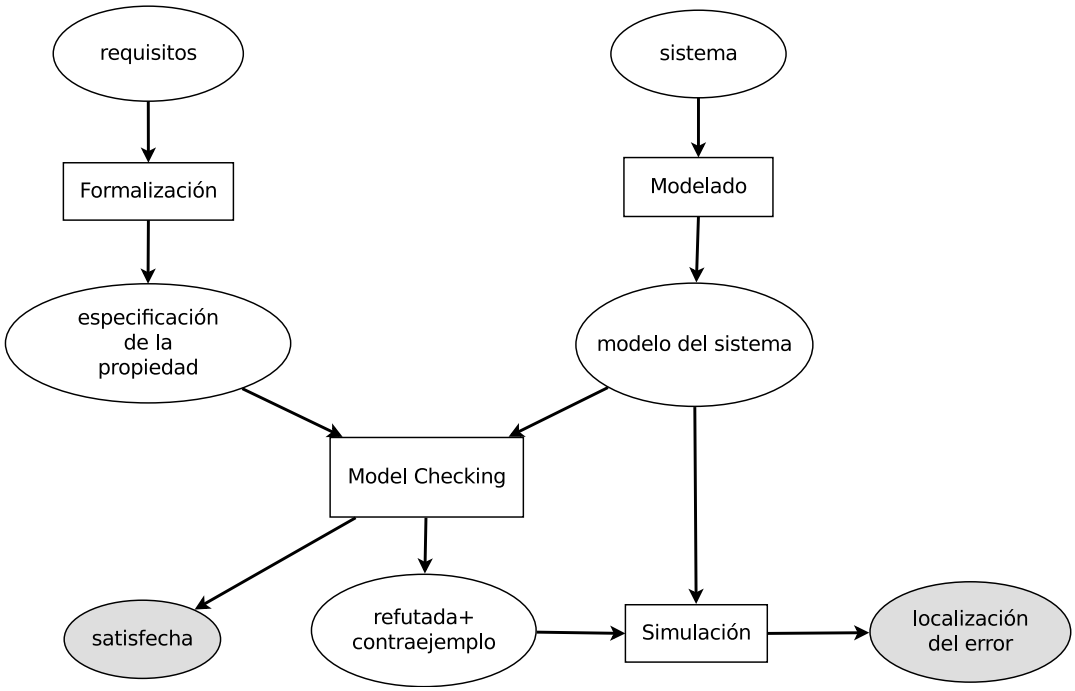

Figura 3.6: El proceso de Model Checking

que todas las propiedades hayan sido validadas por el model checker, lo que indicaría que el modelo del sistema cumple con las características expresadas en la especificación de las propiedades. Por último puede suceder que una o más propiedades hayan sido refutadas. En este caso, la herramienta provee un contraejemplo que nos permite identificar el problema, el cual puede haber sido causado por una mal modelado del sistema, una falla en la especificación, o simplemente porque en efecto el sistema no cumple con la característica deseada.

Es importante tomar en cuenta que el éxito de la verificación depende fuertemente de la corrección en el modelado del sistema y la especificación de las propiedades. El sistema puede tener errores pero haber sido modelado evitando los mismos, lo cual puede llevar a que el model checker entregue falsos positivos. Este efecto también puede ser causa de una especificación incompleta o errónea de las características a verificar.

3.6. Discusión

Veremos más adelante, al momento de presentar las sintaxis de NuSMV y Falluto2.0, como se pone en práctica la teoría presentada en este capítulo. Podremos distinguir por un lado que el lenguaje presentado por Falluto2.0 para modelar los sistemas se acerca en gran medida a la idea de LTS, mientras que por otro lado el de NuSMV será más cercano a la representación de estructuras de Kripke. Esto nos llevará a que la teoría de traducción de LTS a fórmulas de Kripke presentada en la sección 3.1 conforme la base de la semántica de compilación de Falluto2.0.

Este capítulo nos deja entrever así también el gran control que tendremos sobre el sistema final que el model checker verificará, dado que al describir la estructura de Kripke estamos a un paso del armado de los BDD sobres los cuales la herramienta verificará el modelo. La ausencia de pasos intermedios evita la introducción de nuevas variables y la complejización del modelo la cual se traduce en una explosión de estados.

Capítulo 4

NuSMV

4.1. El model checker NuSMV

NuSMV es un model checker simbólico originado en la reingeniería, reimplementación y extensión de SMV, el primer model checker basado en BDD. NuSMV fue diseñado para ser una plataforma bien estructurada, de código abierto, flexible, y ampliamente documentada, para el model checking. Con el fin de permitir que NuSMV sea utilizado en proyectos para la transferencia de tecnologías, fue diseñado para ser muy robusto, cercano a los estándares industriales, y para permitir lenguajes de especificación ampliamente expresivos [16].

NuSMV permite la verificación de propiedades en los formalismos LTL y CTL, sobre modelados finitos de sistemas concurrentes, bajo condiciones de fairness.

Para este trabajo usamos como back-end el model checker *NuSMV2* [19], el cual integra la verificación en base a *BDD* junto con la verificación acotada de propiedades LTL mediante *SAT-Solving*. Otra particularidad de esta versión es el uso de una licencia *OpenSource* para programación y distribución abierta de su código, lo cual permite, a cualquier persona interesada, usar la herramienta gratuitamente y colaborar con el desarrollo de la misma.

4.2. Lenguaje de modelado de NuSMV

Si bien el lenguaje de modelado de NuSMV es amplio y ofrece diferentes posibilidades para la definición de la maquina de estados, presentaremos en

esta sección solo aquella porción que nos servirá más adelante para compilar los modelos descriptos en el lenguaje de Falluto2.0.

Cada proceso del sistema se describe en NuSMV como un módulo, introducido por la palabra `MODULE`. Al menos uno de estos módulos debe llamarse *main*, y será el módulo evaluado por el interprete. Podemos ver cada proceso como una maquina de estados independiente y al conjunto de estos y sus interacciones como la maquina de estados que representa al sistema completo. Veamos entonces como definir la maquina de estado de cada proceso:

Variables

Dentro de cada *módulo*, introduciendo la sección de variables por la palabra `VAR`, definimos las variables del proceso junto con sus dominios. Poseemos tres tipos básicos de variables, las booleanas, las enteras y las enumeradas. Tenemos la posibilidad de declarar una variable como un arreglo de valores de cualquiera de los tipos anteriormente mencionados. Vemos un ejemplo a continuación:

```
MODULE main
    VAR
        var1 : boolean;
        var2 : -2..3;
        var3 : {a1, something, 42};
        var4 : array -1..2 of boolean;
```

Vemos que en el ejemplo la variable `var1` es de tipo booleano (su dominio está formado por los valores `TRUE` y `FALSE`) , `var2` es de tipo entero y su dominio se extiende desde el numero -2 al 3 y `var3` es de tipo enumerado y su dominio lo forman las 3 palabras encerradas en llaves. Por último `var4` es un vector de cuatro booleanos indexado sobre el rango -1 .. 2. Notemos que las valuaciones sobre las variables representan el estado del proceso y una valuación sobre el conjunto de todas las variables de cada proceso representa el estado del sistema completo, tal como se remarcó en la teoría presentada en el capítulo 3.

Estado inicial

Para definir el conjunto de estados iniciales de cada proceso introducimos una nueva sección dentro del módulo usando la palabra clave `INIT`, y hace-

mos uso de una fórmula proposicional sobre las variables del módulo para restringir el subconjunto de estados que deseamos sea el inicial. Recordemos que según lo visto en la sección 3.3 para cualquier conjunto de estados podemos definir una fórmula tal que solo ellos la cumplan.

```
MODULE auto
    VAR
        estado: {encendido, apagado}
        nafta: 0..3
    INIT
        estado = apagado & nafta = 3
```

Transiciones

La palabra clave `TRANS` en NuSMV permite introducir dentro de los módulos una sección donde definir las transiciones de nuestra maquina de estados. Así como ya hemos planteado anteriormente, las transiciones hablarán sobre dos clases de estados: los estados actuales, y los estados posteriores, de manera de definir la relación sobre los mismos. Dada una variable de estado `x`, tenemos que `next(x)` representa el estado de la variables en el momento inmediatamente posterior al actual. Una fórmula proposicional establece entonces las características de la relación de transición. Las ambigedades en esta fórmula con respecto a la maquina de estados representada introducen una elección aleatoria por parte de NuSMV. Esto es útil para representar no determinismo entre saltos de estados.

```
MODULE auto
    VAR
        estado: {encendido, apagado}
        nafta: 0..3
    INIT
        estado = apagado & nafta = 3
    TRANS
        ( estado = encendido & nafta <= 0 )
        ->
        ( next(estado) = apagado )
```

Notemos que en el ejemplo solo definimos condiciones para el caso en que `estado = encendido` y `nafta <= 0`, caso en el que el auto debería apagarse en el próximo estado. En los casos en que se da `estado = apagado` o `nafta >0` el no-determinismo maneja la elección de un estado posterior.

Restricciones y fairness

NuSMV permite establecer restricciones de fairness sobre la ejecución de nuestro sistema. Esto permite revisar nuestro sistema bajo ciertos supuestos con respecto a su ejecución. Tenemos la posibilidad de definir dos tipos de condiciones de fairness sobre nuestro sistema modelado:

- *Fairness incondicional* Dada una fórmula q sobre el estado del sistema, escribimos `FAIRNESS q` para establecer una la condición de fairness incondicional sobre nuestro sistema modelado. Establece que cierta condición q se cumple frecuentemente durante la ejecución del sistema. En términos de lógicas de tiempo lineal, esto correspondería a restringir la ejecución del sistema a solo aquellas trazas donde se cumpla $\mathsf{G}\,\mathsf{F}\,q$.
- *Strong Fairness* Dadas dos fórmulas p y q sobre el estado del sistema, escribimos `COMPASSION(p,q)` para establecer una condición de fairness fuerte sobre el sistema. Esta condición restringe a la revisión de aquellas ejecuciones del sistema en donde valga que si p se cumple siempre a menudo entonces q se cumple siempre a menudo. En términos de LTL esto correspondería a revisar solo aquellas ejecuciones donde se cumpla $\mathsf{G}\,\mathsf{F}\,p \rightarrow \mathsf{G}\,\mathsf{F}\,q$.

Propiedades y verificación

NuSMV permite verificar tanto propiedades LTL como CTL sobre nuestro sistema. Para cada una de ellas tenemos su respectiva palabra clave y la precedemos con la fórmula que exprese la propiedad a verificar. La verificación de estas propiedades se realizara bajo las condiciones que establecen las restricciones de fairness definidas. A continuación mostramos un ejemplo que ilustra la especificación de las propiedades:

```
MODULE auto
    VAR
        estado: {encendido, apagado}
```

```
        nafta: 0..3
    INIT
        estado = apagado & nafta = 3
    TRANS
        ( estado = encendido & nafta <= 0 )
        ->
        ( next(estado) = apagado )

FAIRNESS nafta > 0

LTLSPEC F nafta = 0
CTLSPEC EG estado = apagado
```

En el ejemplo de arriba podemos encontrar la especificación de una propiedad LTL introducida por la palabra clave `LTLSPEC` y una CTL introducida por la palabra clave `CTLSPEC`. Ambas serán verificadas sobre el sistema modelado pero bajo la condición de fairness dada por `FAIRNESS F nafta = 0`.

Discusión

Si bien el lenguaje de NuSMV presenta una amplia variedad de posibilidades para el modelado del sistema, nosotros solo haremos caso a una pequeña parte de este. Con la porción del lenguaje presentado en este capítulo será suficiente para describir la maquina de estados de nuestros modelos compilados de manera completa. Nos permitirá además mantener un claro control sobre el tamaño de nuestro sistema al momento de compilar lo modelado por el usuario en el lenguaje de front-end que presenta Falluto2.0 al lenguaje del back-end ofrecido por NuSMV. Notemos lo cercano que están las descripciones dadas como ejemplo en este capítulo a la idea intuitiva de una estructura de Kripke.

Capítulo 5

La sintaxis de Falluto2.0

En este capítulo repasaremos de manera detallada la sintaxis de Falluto2.0. Comenzaremos explicando como modelar el comportamiento operacional de nuestro sistema. A continuación mostraremos el mecanismo para la inyección de fallas sobre este sistema, y concluiremos describiendo la sintaxis para la especificación de restricciones y propiedades a verificar. Para una referencia aún más completa y precisa sobre las reglas sintácticas, en el apéndice B de este mismo trabajo se puede encontrar una descripción de la sintaxis en términos de Parsing Expresion Grammars.

5.1. Lenguaje de modelado de comportamiento operacional

Para definir el comportamiento de cada proceso involucrado en el funcionamiento de nuestro sistema, usaremos los *proctypes* de Falluto2.0. Cada *proctype* define el funcionamiento interno de una clase de proceso en su totalidad, y los procesos reales pueden ser instanciados a partir del mismo. La declaración de cada *proctype* esta delimitada por las palabras claves `PROCTYPE` y `ENDPROCTYPE`, y en ella podemos distinguir los siguientes elementos:

- El encabezado.
- La declaración de variables de estado.
- La declaración de transiciones de falla.

- La sección de inicialización.
- La sección de declaración de transiciones normales.

A continuación buscaremos comprender el uso de cada una de estas secciones comenzando por el encabezado y mirando luego el cuerpo del proctype, el cual esta conformado por el resto de las secciones enumeradas arriba.

Formando el *encabezado* del proctype encontraremos, después de la palabra clave introductoria `PROCTYPE`, el nombre que denotará a cada clase de procesos. A continuación una lista de nombres especifica si estos procesos poseerán o no variables de contexto, y acciones de sincronización. A modo de ejemplo miremos el encabezado que sigue:

```
PROCTYPE miProceso (ctxVar1, ctxVar2 ; sinchroAct1, sinchroAct2)
```

En este encabezado estamos indicando que los procesos de tipo `miProceso` poseerán dos variables de contexto (`ctxVar1` y `ctxVar2`) y dos acciones de sincronización (`sinchroAct1` y `sinchroAct2`).

En el *cuerpo* del proctype podemos distinguir tres secciones que definen su comportamiento operacional. Las describimos a continuación:

1. *Declaración de variables de estado.* Introducido por la palabra clave `VAR`, esta sección se utiliza para declarar las variables que representan el estado del proceso. Semejante a la sintaxis de NuSMV, podemos declarar tres tipos de variables, sumado a la posibilidad de poder declarar vectores de variables de un tipo determinado. Los siguientes tipos de variables estan disponibles:

   a) *Booleano.* Corresponden a aquellas variables que solo toman los valores `TRUE` y `FALSE`. Las declaramos con la siguiente sintaxis:

   ```
   nombre_de_variable : bool
   ```

   b) *Entero.* A las variables de tipo entero las declaramos dándoles un nombre y acompañándolas con la especificación del rango entero sobre el cual pueden tomar valor. Por ejemplo una variable declarada como sigue, solo podrá tomar valores enteros entre -5 y 7:

   ```
   nombre_de_variable : -5..7
   ```

c) *Enumerado.* Corresponden a las variables de tipo enumerado. Los valores que toman estas variables son definidos como un conjunto de palabras elegidas por el usuario. La sintaxis declarativa de este tipo de variables se asemeja a la del ejemplo que sigue:

```
nombre_de_variable : {a, casa, 34, a2}
```

d) *Vector.* Podemos declarar variables como vectores de alguno de los tipos anteriormente mencionados. De este modo podemos declarar vectores de booleanos, o vectores de enteros o de tipo enumerado. Se debe indicar en la declaración de los vectores el rango de indexación que se desea para ellos. De esta manera no solo se define el tamaño del vector, sino que también se indica qué índices serán permitidos usar para acceder a sus elementos. El ejemplo que sigue es un vector de valores de tipo enumerado, se indexa sobre el rango entero -1 .. 2, y su dominio pertenece al conjunto enumerado {`a,b,c`}:

```
nombre_de_variable : array -1..2 of {a,b,c}
```

2. *Declaración de estados iniciales.* Esta sección es introducida por la palabra clave `INIT`, y define el conjunto de estados iniciales de los procesos de este tipo mediante una fórmula proposicional sobre las variables de estado del proceso. Similar a la sección *INIT* de los módulos de NuSMV, la fórmula que se describe aquí busca restringir o no los valores que puede adoptar cada variables de estado en el estado inicial de la ejecución del proceso. Se permite aquí el uso de conectivos lógicos y proposicionales como $\rightarrow, |, \&, \leftrightarrow, etc$, además de conectivos matemáticos, valores booleanos y otros elementos. La restricción más importante es que la fórmula final sea de tipo booleano.

3. *Definición de las transiciones.* Para definir la relación de transición de nuestros procesos, tomaremos en cuenta cada transición por separado. Podremos etiquetar las transiciones con un nombre, y de esta manera poder referir a la misma al momento de declarar las propiedades a ser verificadas. Llamaremos acción al nombre de la transición. La sintaxis general para la declaración de una transición es la que sigue:

```
[accion] condicion_de_habilitacion => postcondicion ;
```

Como ya dijimos `accion` es el nombre de la transición (representa a la acción que lleva a cabo el sistema para pasar a un nuevo estado), la `condicion_de_habilitacion` es una fórmula booleana sobre el estado actual del sistema, que restringe los estados de partida de la transición a aquellos que la cumplan. Por último, `postcondicion` es un lista de asignaciones a las variables en el estado de llegada. Estas asignaciones definen las características de los estados de destino de esta transición. Para indicar que estamos asignando un valor a una variable en el próximo estado usamos el apostrofe al final del nombre de la variable, por ejemplo si x es una variable en el estado actual entonces x' es una variables en el estado inmediatamente próximo. El siguiente es un ejemplo un poco mas claro para la declaración de una transición:

```
[transicion1] x > 3 & y => x' = x - 1, y' = !y;
```

Vemos en el ejemplo que la precondición restringe los estados origen a aquellos en los que el valor de verdad de `y` sea *TRUE* y el valor numérico de `x` sea mayor a 3. Por su parte se restrige a los estados de llegada al subconjunto de estados que cumplan con la condición de que el valor de `x` sea su valor actual menos 1 y el de `y` sea la negación de su valor actual.

Una vez descripto el funcionamiento de los procesos dentro de cada proctype, podemos instanciar una o mas veces cada uno de ellos para en efecto definir los procesos involucrados en el sistema. Usamos para ello la palabra clave `INSTANCE` y pasamos a la instanciación los parámetros pertinentes, según se haya definido en el proctype correspondiente. La sintaxis sería entonces la siguiente:

```
INSTANCE nombre = nombre_proctype(param1, param2, ..., paramN)
```

Como ya hemos mencionado, podemos pasar variables de contexto como parámetros a nuestras instancias. Estas variables de contexto pueden ser referencias a instancias, o a variables de otras instancias. Para pasar una instancia completa como variable de contexto simplemente colocamos su nombre en el parámetro correspondiente. Para referir a una variable en particular de alguna instancia colocamos el nombre de la instancia precedido por un punto y el nombre de la variable en cuestión:

```
nombre_de_instancia.nombre_de_variable
```

En cuanto a los parámetros de acciones de sincronización, basta con pasar un nombre a elección al parámetro. Todas las acciones con el mismo nombre a nivel de instanciación, serán sincronizadas entre si. Es decir que si quiero sincronizar acciones entre distintas instancias debo tener presente estas acciones parametrizadas en el proctype, y debo otorgarles un mismo nombre en la instanciación de los procesos involucrados. Observemos un ejemplo:

```
PROCTYPE contador ( ; contar )
    VAR
        cuenta:0..5
    INIT
        cuenta = 0
    TRANS
        [contar]: => cuenta' = (cuenta+1)%6;
ENDPROCTYPE

INSTANCE contador1 = contardor(contar1)
INSTANCE contador2 = contardor(contar2)
INSTANCE contador3 = contardor(contar2)
INSTANCE contador4 = contardor(contar1)
```

En este ejemplo hemos instanciado cuatro procesos de tipo *contador*. Si miramos la declaración del proctype `contador` veremos que posee un acción de sincronización denominada `contar`. En el momento de instanciación hemos logrado entonces que el proceso `contador1` sincronice esta acción con `contador4` otorgándoles a ambos el nombre de sincronización `contar1`, mientras que `contador2` sincronizará con `contador3` a partir del nombre de sincronización `contar2`.

5.2. Lenguaje de descripción de fallas

En Falluto2.0 podemos inyectar fácilmente fallas en los procesos modelados. Las fallas forman parte de cada proceso y por lo tanto son declaradas dentro de los proctypes en una subsección introducida por la palabra clave `FAULT`. La sintaxis general para la declaración de una falla es la siguiente:

```
nombre : condicion_habilitacion => postcondicion is tipo
```

Como vemos cada falla tiene un nombre, y su sintaxis es similar a la de una transición común. Esto se debe a que se toma en cuenta la falla como una transición más, aunque no deseada, del sistema. El nombre le otorga a la falla una identificación única dentro del proctype. La condición de habilitación es nuevamente una fórmula booleana sobre el estado actual del proceso (y el sistema global si tomamos en cuenta el uso de las variables de contexto). La postcondición establece condiciones a cumplirse en el estado de llegada luego de la transición de falla. De nuevo, se introduce aquí una lista de asignaciones a las variables de estado a ser cumplidas por el estado de llegada. Las fallas pueden pertenecer a tres tipos en particular: *transient, stop, byzantine.* Podemos distinguir dos clases de fallas según la prolongación de su actividad. Las fallas transient pertenecen a una clase de fallas en donde el efecto es instantaneo, mientras que las fallas de tipo stop y byzantine pertenecen a una clase de fallas de efecto permanente. A continuación daremos la sintaxis para declarar cada uno de estos tipos para las fallas:

- *Tipo Transient.* Simplemente usamos la palabra clave `TRANSIENT` para declarar que la falla pertenece a este tipo. Notar que los efectos de estas fallas se restringen a la postcondición que se haya definido en la declaración de la misma. Ejemplo:

  ```
  falla1 : TRUE => x' = 0 is TRANSIENT
  ```

- *Tipo Stop.* Declara que la falla tiene la capacidad de detener total o parcialmente el funcionamiento del proceso, impidiendo que el mismo realice ciertas transiciones. Usamos la palabra clave `STOP` y opcionalmente la precedemos de una lista con los nombres de transiciones que esta falla, en el caso de ocurrir, inhabilita. Notemos que si optamos por no otorgar dicha lista, la falla detendrá todas las transiciones normales del sistema, entendiendo por normales a aquellas que no son transición de falla. Ejemplo:

  ```
  falla1 : x => y' = FALSE is STOP(trans1,trans2)
  ```

- *Tipo Byzantine.* Declara que la falla provoca un efecto bizantino sobre ciertas variables del proceso. Es decir que una vez ocurrida la falla las variables mencionadas podrán en cualquier momento cambiar su valor actual por algún otro valor dentro de su dominio, provocando asi un

cambio de estado en el sistema. Debemos preceder a la palabra clave `BYZ` con una lista de nombres de variables que serán afectadas. Ejemplo:

```
falla1: TRUE => ... is BYZ(var1, var2)
```

5.3. Lenguaje de especificación de propiedades y restricciones

Con el fin de poder especificar propiedades a cerca de nuestro sistema para su posterior verificación, Falluto2.0 nos ofrece la posibilidad de trabajar sobre los lenguajes de especificación LTL y CTL. Se propone también el uso de meta-propiedades para razonar sobre el comportamiento del sistema bajo los efectos de fallas. Para la especificación de una propiedad LTL usamos la palabra clave `LTLSPEC` y la precedemos de una fórmula LTL que defina la propiedad deseada sobre el total del sistema. Podemos usar para ello operadores LTL comunes como `G`(Globaly), `F`(Finaly), etc, además de los conectivos y operadores de la lógica proposicional. Similarmente para definir propiedades en lógica CTL usamos la palabra clave `CTLSPEC` precedida por la fórmula correspondiente. La sintaxis completa para la formación de las fórmulas puede encontrarse en el apéndice B.

Falluto2.0 posee además las siguientes meta-propiedades para la verificación del sistema:

- *Normal Behaiviour* Usamos esta meta-propiedad para verificar propiedades bajo la premisa de que no ocurren fallas durante la ejecución del sistema. Usamos la palabra clave `NORMAL_BEHAVIOUR` y la precedemos por una fórmula en el formalismo LTL o CTL especificando la propiedad deseada. Luego la propiedad será satisfecha por el sistema si el mismo la satisface en todas aquellas ejecuciones donde no ocurren fallas. La sintaxis es entonces:

  ```
  NORMAL_BEHAVIOUR -> q
  ```

  donde `q` es una fórmula expresada en los formalismos LTL o CTL.

- *Finitely Many Fault/s* Utilizando la sintaxis

  ```
  FINITELY_MANY_FAULT(fault1, fault2, ..., faultn) -> q
  ```

podemos verificar si la propiedad `q` se cumple bajo la suposición de que eventualmente dejan de ocurrir las fallas de nombre `fault1, fault2, ..., faultn` en la ejecución del sistema. En este caso la propiedad `q` solo puede estar escrita en LTL debido a restricciones intrínsecas de la lógica CTL. De manera similar el uso de la construcción

```
FINITELY_MANY_FAULTS -> q
```

permite verificar si la propiedad `q` se cumple bajo la suposición de que finalmente deja de ocurrir falla alguna en el sistema, y el mismo pasa a transitar por transiciones normales.

- *Deadlock check* La utilización de la palabra clave `CHECK_DEADLOCK` dentro de la sección de opciones en la especificación de nuestro sistema, ordena la verificación de que nuestro sistema no caiga en deadlock. Es decir que con esta palabra clave podemos verificar si en algún momento el sistema se ve impedido a realizar transiciones normales, y debe optar por detenerse o realizar transiciones de falla.

Podemos restringir la verificación de las propiedades sobre nuestro sistema utilizando condiciones de fairness. Para ello poseemos las dos siguientes construcciones:

- *Fairness incondicional.* Para restringir la verificación del sistema a solo aquellas ejecuciones donde una propiedad `q` se cumpla siempre a menudo, usamos la siguiente sintaxis:

  ```
  FAIRNESS q
  ```

  donde `q` es una fórmula proposicional que habla a cerca del estado del sistema.

- *Fairnes fuerte.* Dadas dos fórmulas proposicionales `p` y `q` sobre nuestro sistema, podemos restringir la verificación de propiedades LTL a ejecuciones en las cuales se cumpla la condición de justicia fuerte de `p` con respecto a `q` usando la siguiente sintaxis:

  ```
  COMPASSION(p,q)
  ```

De esta manera restringimos la verificación solo a ejecuciones donde se cumpla que si `p` vale siempre a menudo, entonces `q` vale siempre a menudo. Tomemos en cuenta que el uso de esta construcción no es compatible con propiedades definidas en CTL y puede dar resultados erróneos.

En el análisis de propiedades sobre nuestro sistema, no solo podemos hablar sobre el estado del sistema en cuanto a las variables de estado, sino que también podemos analizar las acciones que nos llevan a esos estados. La construcción `just(accion)` nos permite hablar a cerca de la ocurrencia de la transición etiquetada con el nombre `accion`. Por ejemplo podemos restringir nuestra verificación a aquellas ejecuciones en donde la transición etiquetada como `acción1` se lleve a cabo siempre a menudo usando la siguiente restricción de fairness:

```
FAIRNESS just(accion1)
```

De similar manera, usando alguna de las siguientes declaraciones, podemos verificar si en toda ejecución del sistema aquella acción se realiza en algún momento de la ejecución:

```
CTLSPEC AF just(accion1)
LTLSPEC F just(accion1)
```

Vale aclarar que `just(x)`, donde `x` es una acción cualquiera, se toma como una fórmula booleana y puede combinarse dentro de otras fórmulas sin restricción alguna al momento de especificar propiedades y realizar restricciones de fairness.

5.4. Discusión

Hemos presentado un lenguaje que nos permite describir de manera intuitiva el comportamiendo del sistema. Es fácil distinguir como nuestro lenguaje representa de manera directa las maquinas de estado utilizando conceptos derivados de los formalismos como el LTS y estructuras de Kripke, revisados ambos en el capítulo 3 de este trabajo. Vemos como por ejemplo hemos tomado las etiquetas del formalismo LTS para dar nombre a nuestras transiciones. Vemos también como hemos distinguido entre los valores de las variables de estado en el estado actual y en el estado que sigue, tal como se plantea en

las transiciones de una estructura de Kripke. En la sección *INIT* de nuestros *proctypes* podemos ver claramente la especificación de los estados iniciales de las maquinas de estados que representan nuestros procesos.

Es muy importante también el carácter declarativo que se ha dado al lenguaje de modelado de fallas. Esta característica permite una inyección limpia de las mismas en el funcionamiento del sistema, dejando que nuestra herramienta se encargue del aspecto funcional de aquellas. De esta manera evitamos en gran medida la aparición de errores debido a una mala inyección de fallas por parte del usuario, como así también mantenemos control sobre la explosión de estados que puede acarrear el uso de variables innecesarias en lo que podría ser una inyección de fallas ineficiente. Evitamos por último gran cantidad de trabajo al usuario, eliminando la aparición de código correspondiente a la ocurrencia de fallas en las demás secciones del modelado del sistema, y permitiendo que este se concentre en otros aspectos de la funcionalidad. De este modo el usuario podrá una vez declaradas rápidamente las fallas concentrarse por ejemplo en los mecanísmos de tolerancia a las mismas sin involucrarse ni prestar mayor atención a la funcionalidad de las fallas.

Capítulo 6

Semántica de Falluto2.0

Como ya mencionamos en la introducción a este trabajo, Falluto2.0 es un front-end para NuSMV. Por lo tanto la descripción del sistema en el lenguaje de Falluto2.0 debe ser compilada a una descripción en el lenguaje de NuSMV, quien será el encargado de realizar la verificación solicitada sobre el modelo de sistema. En este capítulo abordaremos el tema de la semántica de Falluto2.0, repasando las ideas detrás de su implementación y dando una mirada a la compilación al lenguaje de NuSMV.

6.1. Construcción de un sistema de procesos concurrentes

En la estructura de la especificación del sistema en Falluto2.0 podemos distinguir por un lado la definición del comportamiento de los procesos, llevado a cabo a partir de los proctypes, y por otro lado la instanciación de los diferentes procesos y la especificación de la interacción entre ellos. Se busca que los procesos se desarrollen de manera concurrente en el entorno del sistema global, donde cada uno pueda tener acceso a cierta información del sistema según lo defina el usuario. Se busca así también que los procesos puedan sincronizar acciones y que se pueda controlar de manera general el interleaving del resto de las acciones.

La representación del sistema completo se logra en la compilación a partir del uso de un solo módulo de NuSMV con el siguiente aspecto:

```
MODULE main
```

```
VAR
...
INIT
...
TRANS
...
```

A continuación describiremos como se construye cada sección de este módulo con el fin de lograr la compilación del sistema original descripto usando el lenguaje de Falluto2.0.

Variables del sistema

Podemos ver el estado del sistema como aquel formado por la conjunción del estado de cada proceso que lo compone. De este modo las variables de estado del sistema compilado serán la unión de todas las variables de estado de cada instancia declarada en el sistema original. Por lo tanto en la sección `VAR` de nuestro módulo compilado colocaremos por cada variables de cada instancia del sistema original, una variables en representación. La univocidad de las mismas estará dada por la univocidad del nombre de cada instancia con respecto a sus pares. Otras variables formarán parte de esta sección también:

- *Variable de acción.* Esta variables nos permitirá denotar cual fue la última acción llevada a cabo por el sistema. Su utilidad sin embargo va mas allá de una simple denotación, ya que nos permitirá especificar sobre las acciones llevadas a cabo por el sistema. A partir de esta variable nos será posible manejar la compilación en aquellas situaciones en las que necesitemos hablar sobre propiedades o establecer restricciones sobre el accionar de nuestro sistema. El dominio de esta variables consta de palabras denotando la realización de cada acción del sistema, entre ellas las acciones comunes de cada instancia de proceso, las acciones de falla de cada instancia de proceso, y acciones de efectos de fallas bizantinas. Un último valor en el dominio de esta variable nos permite denotar una transición al estado de deadlock.

- *Variables de activación de fallas.* Las fallas permanentes, es decir las de tipo *stop* y *byzantine*, poseen una variable booleana que nos permite conocer si estas fallas han ocurrido y están afectando el funcionamiento del sistema.

- *Variables de program counter.* Estas variables permiten distinguir entre las acciones de igual nombre dentro de un mismo proceso. Existe entonces un variable de este tipo por cada proceso instanciado. Nos permiten eliminar casos de ambigedad en el salto de estados del sistema, que de otro modo nos llevaría a una representación errónea del sistema original.

Estados iniciales

Como ya dijimos, la restricción de estados iniciales en NuSMV se lleva a cabo a partir de una fórmula booleana dentro de la sección *INIT* del módulo. Nuestra fórmula booleana será entonces la conjunción booleana de todas las fórmulas presentes en la sección *INIT* de cada proceso instanciado. Agregaremos también en esta fórmula condiciones para la inicialización de variables introducidas por la compilación como serían las variables de actividad de falla y las de program counter.

Transiciones

La relación de transición esta representada en la sección *TRANS* del módulo compilado mediante una fórmula booleana que captura cuatro tipos de transiciones sin realizar distinción entre ellas. Nos aseguramos que estas transiciones sean excluyentes de manera de representar correctamente el modelo diseñado en el lenguaje de Falluto2.0. Esta exclusión se lleva a cabo a partir de la variable de acción introducida en el momento de compilación, y en el caso de no ser suficiente con ella, usamos también las variables de program counter (por ejemplo cuando hay mas de una acción con el mismo nombre dentro del mismo proctype). Los cuatro tipos de transiciones corresponden a:

- *Acciones comunes.* Son las transiciones definidas dentro de cada la sección *TRANS* de cada proctype.

- *Acciones de falla.* Son las correspondiente a las transiciones de estados llevadas a cabo a partir de la ocurrencia de las fallas declaradas en la sección *FAULT* de cada proctype.

- *Acciones de efecto bizantino.* Representan el cambio de estado del sistema como consecuencia del accionar de los efectos de una falla bizantina.

- *Acción de deadlock.* Representa la imposibilidad de realizar cualquier acción normal. Decimos entonces que el sistema ha realizado una transición hacia un estado de deadlock. Veremos mas adelante la utilidad de poseer esta transición.

La disyunción excluyente de estas transiciones forma la fórmula de transición final que introducimos en la sección *TRANS* de nuestro módulo compilado, el cual quedará definido en términos de lo que sigue:

MODULE main
VAR
** Por cada instancia de proceso, introducimos la lista*
de variables declaradas en su respectivo proctype.
** Variable de accion.*
** Variables de actividad de fallas permanentes.*
** Variables de program counter por cada proceso instanciado.*
INIT
Conjuncion de fórmulas de la seccion INIT de procesos instanciados
& Inicializacion de variables introducidas en compilacion.
TRANS
disyuncion de transiciones comunes compiladas
| disyuncion de transiciones de falla compiladas
| disyuncion de transiciones de efectos bizantinos
| transicion de deadlock

6.2. Compilación de transiciones

Veremos a continuación como se realiza el proceso de compilación de cada una de las transiciones que luego forman parte de la fórmula de la sección *TRANS* del módulo compilado.

Transiciones normales

Recordemos que la siguiente era la sintaxis general de la declaración de una transición normal en el lenguaje de Falluto2.0 (Para mas detalles referirse al capítulo 5):

$$[nombre]\ habilitacion => postcondicion;$$

La versión compilada de esta acción tendrá el siguiente aspecto:

$$next(actvar) = nombre\ \&\ !faultact\ \&\ habilitacion\ \&\ postcondicion\ \&\ resto$$

Notar que de ocurrir esta transición, el nombre de la misma quedará guardado como valor de actvar en el próximo estado, indicando que, en efecto, ésta fue la acción que se llevo a cabo para llegar a ese estado. *!faultact* es una fórmula que asegura que no estén activas aquellas fallas de tipo STOP que afectan esta transición. En efecto esta fórmula es simplemente la conjunción de la negación de las variables de actividad de falla para aquellas fallas que afectan esta transición. *habilitación* es simplemente una compilación directa de la fórmula de habilitación original al igual que *postcondicion* la cual es la conjunción de la compilación directa de las asignaciones establecidas en la postcondición original. Por último en *resto* nos aseguramos que en el próximo estado no cambie el valor de las variables cuyo valor no es definido en esta transición. En efecto esta fórmula es la conjunción de las asignaciones `next(x)=x` por cada variable `x` que debe mantener su valor en el estado destino. Notemos por último que si la transición se realiza a partir de una acción de sincronización entonces *habilitación* es la conjunción de todas las condiciones de habilitación dadas en cada proceso involucrado. De manera similar la postcondición es la conjunción de todas las asignaciones establecidas en la postcondición de la transición correspondiente en cada proceso involucrado, y se debe tomar en cuenta la inactividad de las fallas de tipo STOP de los diferentes procesos.

Transiciones de falla

Recordemos que en cuanto a nosotros respecta, una falla es una transición más del sistema. Por lo tanto, serán compiladas de manera muy parecida a las transiciones normales, tomando en cuenta algunas características especiales intrínsecas al tipo de falla. Habíamos dicho que para la inyección de una falla introducíamos de manera declarativa las características de la misma usando

la siguiente sintaxis:

$$nombre \ : \ habilitacion \ => \ postcondicion \ is \ tipo$$

La compilación de esta falla dará como resultado la siguiente fórmula booleana:

$$next(actvar) = nombre \ \& \ !fallaactiva \ \& \ next(fallaactiva) \ \&$$

$$habilitacion \ \& \ postcondicion \ \& \ resto$$

Como en el caso de las transiciones normales, el valor de la variable de acción relacionará el estado destino a ésta transición, la cual en este caso es de falla. *!fallaactiva* es la condición que impide que las fallas permanentes ocurran mas de una vez, en efecto esta condición es TRUE si la falla es de tipo *transient*. En el caso de ser falla permanente y de llevarse a cabo esta transición, la activación de la falla quedará registrada en el próximo estado debido a la condición *next(fallaactiva)* presente en la fórmula. El resto de la fórmula se puede intuir de lo redactado en el caso de las transiciones normales.

Transiciones de efectos de fallas bizantinas

Una vez ocurrida una falla bizantina, sus efectos comienzan a suceder espontáneamente a lo largo de toda le ejecución del sistema. Las variables afectadas por estas fallas cambiarán aleatoriamente de valor en distintos instantes de tiempo durante la ejecución. Para simular este efecto, es que hemos introducido transiciones especiales que describen estos cambios. Una vez ocurrida la falla, su transición de efecto bizantina queda habilitada y puede ocurrir aleatoriamente en cualquier momento posterior. La compilación entonces de estas transiciones resulta en fórmulas semejantes a la que sigue:

$$fallaactiva \ \& \ next(actvar) = bizefect \ \& \ permanecen$$

Aquí *fallaactiva* es la condición de habilitación de estas transiciones, las cuales solo pueden ocurrir si la falla bizantina ya ha ocurrido en algún momento de la ejecución. Como en el resto de las transiciones, *next(actvar)=bizefect* se encarga de colocar el valor correcto a la variable *actvar* con el fin de indicar la ocurrencia de esta transición. Por último *permanecen* se forma por la conjunción de las fórmulas `next(x)=x` para todo `x` variable del sistema excepto por `actvar` y por aquellas variables afectadas por la falla bizantina. NuSMV entonces elegirá un valor aleatorio para estas variables afectadas ya

que no encuentra restricción alguna en la fórmula. En efecto solo restringimos los valores de ciertas variables y de esta manera definimos un conjunto de estados destino posibles.

Transición de deadlock

Consideramos que nuestro sistema cae en situación de *deadlock* cuando todas sus transiciones normales quedan inhabilitadas. La implementación de una transición que exprese el movimiento del sistema hacia este estado nos es útil para la verificación de diferentes propiedades y para la implementación de ciertas restricciones de fairness. La compilación de la transición de deadlock tiene la siguiente forma:

$$!trans\_habilitadas \ \& \ next(actvar) = deadlock \ \& \ resto$$

Como es evidente, la condición de habilitación de esta transición esta dada por la conjunción de la negación de las condiciones de habilitación de todas las transiciones normales del sistema. De este modo la ejecución del sistema realizará la transición de deadlock solo cuando no este habilitada a realizar ninguna de las transiciones normales del sistema. *next(actvar)=deadlock* se encarga de otorgar valor a la acción que nos lleva al deadlock. En *resto* encontramos las asignaciones correspondientes al estado posterior de cada variable en el sistema excepto *actvar*, asegurando que ninguna de ellas cambie de valor en el próximo estado.

Construcción de la fórmula de transición

Una vez compilada cada una de estas transiciones, las mismas son colocadas en la sección *TRANS* del módulo compilado. Como ya mencionamos, es a partir de las disyunción exclusiva de cada una de ellas, que se construye la fórmula que define la relación de transición de nuestro sistema compilado. Es importante notar el carácter exclusivo de esta disyunción, ya que de no poseer esta característica, estaríamos permitiendo transiciones que llevan a cambios de estados no presentes en el modelado original. En cada momento de la ejecución solo una componente de esta disyunción debe ser satisfecha. Notemos que dado que no hay más de una instancia con el mismo nombre, la disyunción exclusiva se da de manera directa entre acciones de distintas instancias. Esto se debe a que el valor que toma la variable `actvar` representa de

manera univoca a la instancia que esta provocando la transición de estados. Si tomamos en cuenta dos acciones cualquiera de una misma instancia, encontramos que no hay problema si el nombre de las acciones difiere entre una y otra ya que esto se ve reflejado nuevamente en el valor que tomará `actvar`. Sin embargo el valor de `actvar` no nos permite distinguir entre dos acciones de igual nombre dentro de una misma instancia. Este problema nos obliga a introducir algún otro mecanismo para lograr la exclusividad en la disyunción de la fórmula de transiciones. En Falluto2.0 se soluciona esto presentando un valor particular de la variables de *program counter* para cada transición dentro de la instancia. El valor que tome esta variable, junto con el valor que tome la variable `actvar` son entonces suficiente para poder distinguir entre las diferentes transiciones y lograr la disyunción en la fórmula.

6.3. Fairness de fallas y procesos

En general nos interesa trabajar sobre diseños de sistemas en los cuales se den ciertas condiciones de fairness comunes al comportamiento real de los mismos. Por ejemplo dado un sistema en el que están involucrados varios procesos no es común buscar verificar propiedades sobre ejecuciones en las que solo uno de estos procesos sea el que actúe. En el caso de presencia de fallas, tampoco es común atender a ejecuciones en donde las fallas se apoderan de los saltos de estado, y no se da lugar a ejecuciones normales en el sistema. Es por ello que en Falluto2.0 se establecen por defecto dos condiciones de fairness específicas, las cuales por supuesto pueden ser desactivadas.

La primera de ellas es una condición de *fairness incondicional para las acciones normales* del sistema. Esta condición evita prestar atención a aquellas ejecuciones en donde el avance en la ejecución es dado solo por transiciones de falla. Para lograr esto, introducimos en el sistema compilado la siguiente fórmula de justicia incondicional (mas conocida como fairness incondicional):

$$FAIRNESS \bigvee_{t \in T_N} actvar = t_{name} \vee actvar = deadlock$$

Donde T_N es el conjunto de transiciones normales del sistema y t_{name} representa el nombre de la transición t. De esta manera estamos pidiendo que durante a ejecución del sistema ocurra que a menudo se realice una transición normal o se caiga en deadlock. Recordemos que para poder caer en deadlock, todas las transiciones normales deben estar inhabilitadas.

Nuestra segunda condición de fairness por defecto se forma a partir de varias condiciones de fairness que aseguran que si un proceso cualquiera esta habilitado siempre a partir de un momento, entonces siempre a menudo sea atendido para realizar una transición normal. Decimos entonces que ésta es una *condición de fairness débil para procesos*. Para implementar esta condición pediremos, por cada proceso, que siempre a menudo o bien el proceso este en deadlock, o bien se lo atienda para realizar alguna acción normal. Definimos que el proceso esta en deadlock si todas sus acciones nomrales están deshabilitadas. Agregamos entonces por defecto la siguiente condición de fairness a nuestro sistema compilado:

$$FAIRNESS \bigvee_{t \in T_{N_{proc}}} actvar = t_{name} \vee proc\_deadlock$$

donde $T_{N_{proc}}$ es el conjunto de transiciones normales del proceso en cuestión, *proc_deadlock* es la fórmula que expresa que este proceso cayó en deadlock y la construimos como sigue:

$$\bigwedge_{t \in T_{N_{proc}}} !habilitacion_t$$

donde $habilitacion_t$ es la condición de habilitación para la transición t. Notemos que un proceso puede caer y salir de deadlock sucesivamente debido a cambios en el estado del resto del sistema. Por lo tanto puede darse el caso de que este fairness se cumpla y sin embargo no se atienda al proceso nunca, ya que cada vez que se lo intenta atender éste haya caído en deadlock. Por lo tanto decimos que la condición de fairness es débil y solo se asegurará atender al proceso si el mismo, a partir de cierto momento de la ejecución, permanece continueamente habilitado para realizar acciones normales.

6.4. Semántica de propiedades

Como vimos al finalizar el capítulo 5, Falluto2.0 permite razonar y verificar sobre las acciones que se llevan a cabo en la ejecución del sistema para realizar los saltos de estado. Recordemos que para referirse a la ocurrencia de una acción del sistema usábamos la construcción:

```
just(accion)
```

Esta construcción permite expresar el nombre de la transición que se ha tomado para llegar al estado actual, en el caso de arriba el nombre de la transición es `accion`. Por lo tanto, la fórmula `just(accion1)` se satisface en un estado cualquiera solo si `accion1` fue la acción realizada para llegar a ese estado. Notemos que esta fórmula nunca se satisface al comienzo de una ejecución, ya que no se realiza ninguna acción para llegar al estado inicial en este momento. La compilación de estas fórmulas se realiza de manera muy simple, verificando en el estado correspondiente, el valor de la variable de acción introducida por Falluto2.0. Es decir que la compilación se reduce a traducir fórmulas de la forma `just(x)` a fórmulas de la forma `actvar = x`.

Compilación de propiedades simples

Falluto permite al usuario definir propiedades en los formalismos LTL y CTL usando la sintaxis `LTLSPEC q` y `CTLSPEC p` donde `q` es una fórmula en el formalismo LTL y `p` una en el formalismo CTL. La compilación de estas fórmulas se hace de manera directa debido a la similitud entre el valor semántico definido por Falluto2.0 y el definido por NuSMV para este tipo de expresiones. Esto vale también para las condiciones de fairness definidas por el usuario con las sintaxis `FAIRNESS q` y `COMPASSION(p,q)`.

Compilación de meta-propiedades

En el caso de las meta-propiedades ofrecidas por Falluto2.0, la compilación no se realiza de manera directa, y por lo tanto conlleva otro tipo de trabajo. A continuación describimos cómo es el proceso de compilación de cada una de ellas:

- *Meta-propiedad Finitely Many Fault/s.* Esta meta-propiedad permite razonar sobre propiedades en el sistema bajo la suposición de que ciertas fallas dejan de ocurrir a partir de cierto momento en la ejecución. La sintaxis que usamos es `FINMANYFAULT(`$\mathtt{f}_1$`,`$\mathtt{f}_2$`,...,`$\mathtt{f}_n$`)` $\rightarrow$ `q` donde $\mathtt{f}_1$`,`$\mathtt{f}_2$`,...,`$\mathtt{f}_n$ son nombres de fallas y `q` es la propiedad LTL que queremos verificar en el sistema. Esta propiedad se compila como una fórmula LTL compuesta a partir de una implicancia, en donde el antecedente sugiere que en cierto memento las faltas mencionadas dejarán de ocurrir, y el precedente corresponde a la compilación directa de la propiedad `q`. El ejemplo quedaría compilado entonces como:

$$\texttt{LTLSPEC (F ! (actvar in f}_1\texttt{,f}_2\texttt{,...,f}_n\texttt{)) } \rightarrow \texttt{q}_{compilada}$$

- *Meta-propiedad Normal Behaiviour.* La meta-propiedad Normal Behaiviour puede ser usada para verificar propiedades bajo la suposición de que la ejecución del sistema estará libre de ocurrencia de fallas. Es decir que es utilizada para verificar si la propiedad deseada se cumple al menos en aquellos casos en los que el sistema avanza solo a partir de transiciones normales. La sintaxis para esta propiedad en el lenguaje de Falluto2.0 es la que sigue:

  ```
  NORMALBEHAIVIOUR ->q
  ```

  donde `q` puede ser una fórmula en formalismo LTL o CTL. La compilación para el caso en que `q` está redactada en LTL es la que sigue:

  $$\texttt{LTLSPEC (G !(actvar in \{f}_1\texttt{, f}_2\texttt{,..., f}_n\texttt{\})) } \rightarrow \texttt{q}_{compilada}$$

  De manera muy similar logramos compilar el caso en que `q` es una fórmula CTL:

  $$\texttt{CTLSPEC (AG !(actvar in \{f}_1\texttt{, f}_2\texttt{,..., f}_n\texttt{\})) } \rightarrow \texttt{q}_{compilada}$$

  En ambos casos $\{\texttt{f}_1\texttt{, f}_2\texttt{,..., f}_n\}$ es el conjunto de nombres de todas las fallas declaradas en el sistema por el usuario.

Falluto2.0 ofrece la posibilidad de verificar que nuestro sistema no caiga en estado de *deadlock*. La instrucción especificada por la sintaxis `CHECKDEADLOCK` busca verificar entonces la ausencia de estados a partir de los cuales no se pueda avanzar usando acciones normales. Es decir que revisa que no hayan estados en los cuales todas las acciones normales estén inhabilitadas. Recordando la semántica de la transición de deadlock presentada en este mismo capítulo, vemos que su condición de habilitación es en efecto la negación de las condiciones de habilitación de cada transición normal del sistema. Gracias a esto es que podemos definir entonces la compilación de nuestra propiedad de ausencia de deadlock como sigue:

```
LTLSPEC G ! actvar = deadlock
```

Luego en presencia de deadlock NuSMV encontrará un contraejemplo en donde en al menos un estado de la ejecución la variable de acción tenga el valor representante de deadlock, sugiriendo por lo tanto que el estado anterior a este poseía inhabilitadas todas las acciones buenas del sistema.

6.5. Discusión

Dado que en el model checking el problema de la explosión de estados es el principal defecto, Falluto2.0 busca en evitar la introducción de complejidad con respecto al sistema original al momento de la compilación. Los mecanísmos implementados para la compilación apuntan a evitar la introducción de nuevas variables, y a su vez de lograr fórmulas de sencillas de verificar. Notemos que las únicas variables introducidas en momento de compilación son prácticamente indispensables. En efecto hemos introducido solo tres tipos de variables en esta etapa:

1. La variable de acción.
2. Las variables de actividad de fallas permanentes.
3. Las variables de program counter.

Cada una de ellas cumple un propósito claro y no hemos encontrado manera más eficiente de lograr este propósito.

Capítulo 7

Casos de estudio

La intención de este capítulo es la de hacer un repaso sobre el uso de Falluto2.0 y sus funcionalidades. Para ello se mostrará el trabajo sobre una serie de problemas de carácter paradigmático en el estudio de la verificación de sistemas. Mostraremos como usar Falluto2.0 en un principio para describir la funcionalidad de estos sistemas. Continuaremos pensando sobre las fallas que pueden afectar los mismos, y buscaremos probar ciertas propiedades de los sistemas descriptos. Finalizaremos cada sección razonando sobre los resultados expuestos por nuestra herramienta.

7.1. Commit atómico de 2 fases (2PC)

El commit atómico es un problema muy conocido en el ámbito de las bases de datos y sistemas de revisión distribuidos entre otros. En estos sistemas es usual encontrar situaciones donde se requiere realizar operaciones de manera atómica, es decir que estas operaciones se realicen de una sola vez, como si se tratase de una única operación. Muchas veces esto requiere de la sincronización y votación de los distintos procesos que componen el sistema.

El protocolo de commit de dos fases (2PC) es un tipo de protocolo de commit atómico (ACP). Es un algoritmo distribuido el cual coordina a todos los procesos participantes en una transacción atómica distribuida por la cual deberán decidir si hacer commit de la transacción o abortar. Es por ende un caso particular de algoritmo de consenso.

Podemos distinguir en este algoritmo a un nodo en particular denominado *coordinador*, el cual estará encargado de iniciar la propuesta de commit, y mas

adelante tomar una decisión a partir de la votación realizada por el resto de los nodos. Los demás nodos serán denominados *votantes*, estos comunmente poseen la información necesaria para realizar la transacción, y serán los que voten por *SI* o por *NO* a la propuesta del *coordinador*.

La especificación del algoritmo busca lo siguiente. Cada proceso votará por *SI* o por *NO*, llegando así a la decisión de commit o a la de abortar, de modo que se cumpla:

1. Si no ocurren errores y todos los procesos votan por *SI*, entonces todos los procesos llegan a la decisión de commit.

2. Un proceso llega a la decisión de commit solo si todos los procesos votaron *SI*.

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

Este algoritmo se ejecuta (como bien dice su nombre) en dos fases: En una primera fase los procesos votaran si se comprometen a realizar el commit. En una segunda fase el coordinador tomara una decisión basada sobre los votos obtenidos, e informará del resultado de su decisión a cada uno de los procesos involucrados en la transacción [17].

El proceso coordinador, de ahora en mas C, posee tres acciones. En la primera acción C emite su voto, pasa a su fase dos, y espera a que los demás procesos voten. En la segundo acción, C detecta que todos los demás procesos han votado *SI* y llega a la decisión de commit. En la tercera acción C detecta que algún proceso ha votado *NO* o se ha detenido, y por lo tanto llega a la decisión de abortar.

Todos los otros procesos, de ahora en mas V, también poseen tres acciones. En su primera acción, V detecta que C ha emitido su voto por commit y emite su propio voto. En su segunda acción V detecta que C se ha detenido y decide abortar. En la tercera acción, V detecta que algún proceso ha llegado a su fase dos, y toma la misma decisión que aquel.

Cada proceso j en nuestro sistema poseerá las siguientes variables de estado:

- $j.p$ Indica la fase del proceso. Comenzará con el valor 0, pasará a tener valor 1 cuando el proceso haya emitido su voto, y finalmente valdrá 2 cuando el proceso haya tomado su decisión final.

- *j.d* Dependiendo de la fase en la que se encuentre el proceso esta variable representará el voto emitido (fase 1), o la decisión tomada (fase 2). El valor *TRUE* indicara voto a favor o decisión de commit, el valor *FALSE* indicará voto en contra, o decisión de abortar.

- *j.up* Esta variable indicará si el proceso está en ejecución (tomando valor *TRUE*) o se ha detenido por algún error (tomando valor *FALSE*).

El siguiente es un diagrama de estados representando a cada uno de los procesos descriptos. Las lineas punteadas representan transiciones de falla en las que provocan que la componente se detenga:

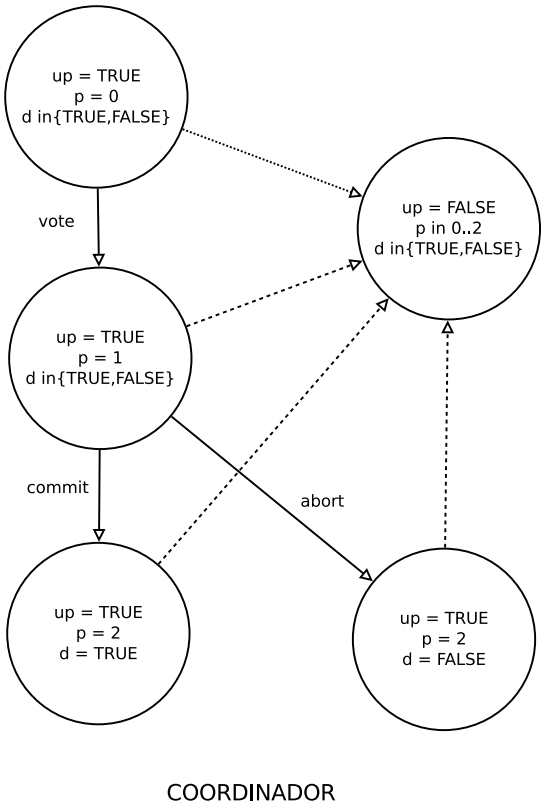

Figura 7.1: Diagrama de estados de una componente coordinador del protocolo de commit atómico de 2 fases

En nuestro caso de estudio no tomaremos en cuenta fallas en la comunicación entre los procesos, solo simularemos fallas de tipo STOP, representando así la caída en inactividad de algún proceso interviniente. Miremos

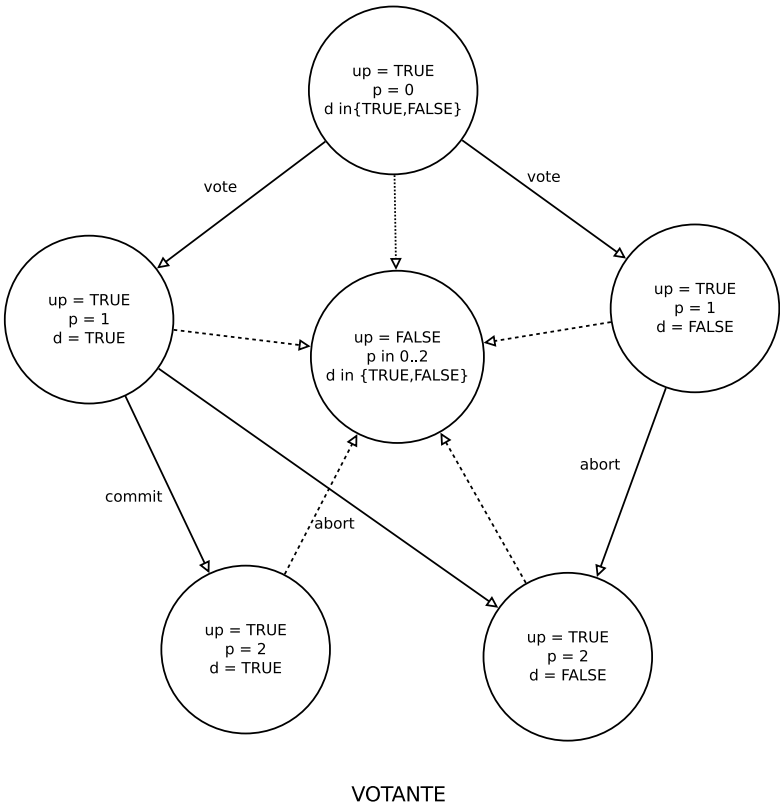

Figura 7.2: Diagrama de estados de una componente votante del protocolo de commit atómico de 2 fases

entonces el modelado de este protocolo en Falluto2.0, tomando el caso de cuatro votantes y un coordinador. Comenzamos modelando la clase de proceso coordinador:

```
PROCTYPE Coordinator(v0, v1, v2, v3)

    VAR
        p   : 0..2
        d   : bool
        up  : bool

    FAULT
        crash: => up' = FALSE is STOP

    INIT
        p = 0 & up = TRUE

    TRANS
        [vote]:    p = 0 => p' = 1, d' in { TRUE, FALSE };
        [commit]:  p = 1 &
                   v0.up & v0.d & v0.p = 1 &
                   v1.up & v1.d & v1.p = 1 &
                   v2.up & v2.d & v2.p = 1 &
                   v3.up & v3.d & v3.p = 1
                   => p' = 2, d' = TRUE;
        [abort]:  p = 1 &
                  ((!v0.up | (v0.p >= 1 & !v0.d)) |
                   (!v1.up | (v1.p >= 1 & !v1.d)) |
                   (!v2.up | (v2.p >= 1 & !v2.d)) |
                   (!v3.up | (v3.p >= 1 & !v3.d)) |
                   !d)
                  => p' = 2, d' = FALSE;
ENDPROCTYPE
```

Como vemos el modelado surge de manera directa a partir de la especificación. Miremos sin embargo con detención la sección *FAULT* del modelado

del proceso coordinador. En ella encontramos la especificación de una única falla, representando la detención del proceso. Esta falla es por ende de tipo *STOP*, y como vimos es de carácter permanente, es decir que una vez que ocurre, el proceso instanciado quedará infinitamente inhabilitado para realizar cualquier transición buena. Dado que es de nuestro interés que los demás procesos puedan enterarse de la ocurrencia de la falla, o mas bien de la detención del proceso coordinador, hemos agregado a la falla de detención la postcondición *up' = FALSE*. Veremos mas adelante como los demás procesos pueden acceder a esta variable, en función de variable de contexto, y de esta manera enterarse de la situación del coordinador. Vale la pena también revisar los parámetros de contexto de esta declaración. Como vemos posee 4 variables de contexto: *– v0, v1, v2, v3–*. Cada una de estas será una referencia a cada votante en el sistema, y serán especificados en la instanciación al realizar el pasaje de parámetros. De este modo el proceso coordinador podrá en cualquier momento acceder a las variables de estado de los votantes en condición de solo lectura.

Continuando con el modelado, miremos ahora los procesos votantes:

```
PROCTYPE RegularVoter(coo, v0, v1, v2)

    VAR
        p:  0..2
        d:  bool
        up: bool

    FAULT
        crash: => up' = FALSE is STOP

    INIT
        p = 0 & up = TRUE

    TRANS
        [vote]: p = 0 & coo.up & coo.p = 1
                =>
                d' in {TRUE, FALSE}, p' = 1;
        [abort]: p = 0 & !coo.up => p' = 2, d' = FALSE;
        [commit]: p < 2 & coo.p = 2 => p' = 2, d' = coo.d;
```

```
        [commit]: p < 2 & v0.p = 2 => p' = 2, d' = v0.d;
        [commit]: p < 2 & v1.p = 2 => p' = 2, d' = v1.d;
        [commit]: p < 2 & v2.p = 2 => p' = 2, d' = v2.d;

ENDPROCTYPE
```

Nuevamente el modelado se extrae de manera directa de la especificación del algoritmo. Vemos como cada proceso tiene acceso en forma de solo lectura a cada uno de los demás procesos en el sistema, incluyendo al coordinador (*coo*). La falla declarada es idéntica a la declarada en el modelado del coordinador. Por último la acción *commit* fue separada para cada caso en que se detecta la toma de decisión de un proceso diferente. Si bien comparten el nombre, solo una de ellas será elegida para realizar la transición.

Miremos ahora como organizar los procesos en la etapa de instanciación de manera que cada uno pueda ver el estado de los demás. Para instanciar los procesos usaremos el siguiente código:

```
INSTANCE coord = Coordinator(voter0, voter1, voter2, voter3)
INSTANCE voter0 = RegularVoter(coord, voter1, voter2, voter3)
INSTANCE voter1 = RegularVoter(coord, voter0, voter2, voter3)
INSTANCE voter2 = RegularVoter(coord, voter0, voter1, voter3)
INSTANCE voter3 = RegularVoter(coord, voter0, voter2, voter1)
```

Como vemos, hemos instanciado un proceso coordinador y cuatro procesos votantes. Hemos asignado los nombres de instanciación de cada proceso como parámetros de los demás procesos en el orden correspondiente de manera que tenga sentido el modelado de los proctypes. Esto queda claro en el caso de los procesos votantes, donde el índice del parámetro para el proceso coordinador en la instanciación coincide con el índice de la variable de contexto *coo* en el proctype *RegularVoter*.

Una vez modelado el sistema, es necesario especificar las propiedades deseadas sobre el mismo, para su posterior verificación con la herramienta. Vimos en la especificación del algoritmo que estamos trabajando, que buscamos que el mismo cumpla con tres propiedades especificas. Las recordamos a continuación y miramos el código usado para especificarlas en Falluto2.0:

1. *Si no ocurren errores y todos los procesos votan por SI, entonces todos los procesos llegan a la decisión de commit.*
   Esta propiedad supone que no ocurren errores y en ese contexto busca probar que si todos los procesos votaron por hacer commit entonces todos llegan a la decisión de hacer commit. Podemos aprovechar entonces la metapropiedad *NORMAL_BEHAIVIOUR* para este caso, y la especificación en Falluto2.0 tomará entonces la siguiente forma:

```
NORMAL_BEHAIVIOUR -> G ( (coord.p = 1 & coord.d &
                          voter0.p = 1 & voter0.d &
                          voter1.p = 1 & voter1.d &
                          voter2.p = 1 & voter2.d &
                          voter3.p = 1 & voter3.d)
                          ->
                          F (coord.p = 2 & coord.d &
                             voter0.p = 2 & voter0.d &
                             voter1.p = 2 & voter1.d &
                             voter2.p = 2 & voter2.d &
                             voter3.p = 2 & voter3.d)
                        )
```

---

2. *Un proceso llega a la decisión de commit solo si todos los procesos votaron SI.*
   Notemos que dado un proceso j podemos distinguir los estados en los que este proceso llega a la decisión de commit a partir de las condiciones $j.p = 2$ y $j.d = TRUE$. Así también, sabemos que si un proceso j se encuentra en una fase que no sea *0*, y su variable d tiene el valor booleano *TRUE* entonces se puede decir que el proceso a votado a favor del commit. De este modo la especificación de esta propiedad en Falluto2.0 toma la forma:

```
LTLSPEC G ( ((coord.p = 2 & coord.d) |
             (voter0.p = 2 & voter0.d) |
```

```
    (voter1.p = 2 & voter1.d) |
    (voter2.p = 2 & voter2.d) |
    (voter3.p = 2 & voter3.d) |
   )
   ->
   (voter0.p != 0 & voter0.d &
    voter1.p != 0 & voter1.d &
    voter2.p != 0 & voter2.d &
    voter3.p != 0 & voter3.d
   )
)
```

---

3. *Todos los procesos que llegan a una decisión, llegan a la misma.* Miremos esta propiedad en particular para los procesos *voter1* y *voter3*, ya que los demás casos son semejantes. Supongamos que cada uno ha llegado a una decisión y veamos que sea la misma. La siguiente sintaxis especifica esta propiedad como una propiedad en CTL:

```
CTLSPEC AG ( (voter1.p = 2 & voter3.p = 2)
             ->
             (voter1.d = voter3.d)
           )
```

Si verificamos con Falluto2.0 el modelo que hemos ido definiendo, junto con las propiedades especificadas, notaremos que en efecto el modelo las cumple. Falluto2.0 expondrá, por cada propiedad presentada, una linea asegurando las satisfactibilidad de la misma en el modelo.

7.2. Protocolo de transmisión satelital

Modelaremos en esta sección, un protocolo de comunicación entre satélites. El protocolo presentado a continuación pertenece a la familia de protocolos de

ventana deslizante con *GO-BACK-N*. Sin embargo distinguimos en el mismo algunas cualidades no muy comunes en estos protocolos. Una de estas diferencias se observa en nuestro sistema de retransmisión el cual no se basa en timeouts de tramas, si no que es instantáneo y constante. Es decir que nuestro emisor retransmitirá constantemente tramas cada vez que este en estado ocioso y posea tramas aún no confirmadas en su ventana de envío.

Los *protocolos de ventana deslizante* pertenecen al conjunto de protocolos de transmisión de datos con control de error (ARQ), comúnmente presentes en los algoritmos de capas de enlace de datos. Estos protocolos se caracterizan por el uso de *números de secuencia* en sus tramas, y el envío de *mensajes de confirmación de recepción*.

En los *protocolos de ventana deslizante*, en cualquier instante, el emisor mantiene un grupo de de números de secuencia que corresponde a las tramas que tiene permitido enviar. Se dice que estas tramas caen dentro de la ventana emisora. De manera semejante, el receptor mantiene una ventana receptora correspondiente al grupo de tramas que tiene permitido aceptar. Los números de secuencia en la ventana del emisor representan tramas enviadas, o que pueden ser enviadas, pero cuya recepción aún no ha sido confirmada. Una vez que se recibe la confirmación de alguna trama en la ventana, se hace lugar en la misma para el envío de una nueva trama. La ventana receptora corresponde, como ya dijimos, a las tramas que puede aceptar. Toda trama que caiga fuera de la ventana se descarta sin más. Cuando se recibe una trama cuyo número de secuencia se corresponde con el límite inferior de la ventana, esta trama se acepta, se genera un mensaje de confirmación de recepción, y la ventana avanza para permitir la recepción de una nueva trama de ser necesario. De este evento surge el nombre de este grupo de protocolos. Las ventanas de recepción conservan su tamaño inicial siempre [**?**, tanembaum]

El protocolo *GO-BACK-N* es un caso específico de protocolos de ventana deslizante, en donde el *tamaño de la ventana de recepción es 1*. Este protocolo se caracteriza por la retransmisión de tramas en tiempos en que el emisor en otro caso quedaría ocioso. En el caso particular del protocolo que modelaremos aquí, si el emisor ha enviado todas las tramas de su ventana y no ha recibido confirmación de recepción para ninguna de ellas, entonces procederá a retransmitir todas las tramas desde la última con confirmación de recepción. Esta característica representa un uso mas eficiente de la conexión con respecto a los protocolos de detención y espera, ya que por un lado el emisor no debe esperar confirmación de recepción para seguir enviando tramas dentro de su ventana, y por otro lado en el caso de pérdidas de tramas,

este se anticipa al reenvío de las mismos. Este tipo de protocolos es muy útil en situaciones en las que retardo en los canales de transmisión es muy grande, ya que en el caso de no retransmitir los tiempos de ocio del emisor serían muy elevados y poco aprovechados.

Con la intención de modelar el algoritmo que especifica este protocolo, dividiremos el mismo en cuatro partes:

1. Envío de trama de secuencia
2. Recepción de trama de secuencia
3. Envío de tramas de confirmación de recepción
4. Recepción de tramas de confirmación de recepción

Modelaremos el módulo receptor de un satélite, y el módulo emisor de otro satélite, los cuales se comunicarán usando el protocolo. El emisor utilizará entonces los algoritmos (1) y (4) mientras que el receptor los algoritmos (2) y (3). A continuación describiremos cada uno de estos algoritmos y mostraremos como modelarlos con Falluto2.0. Nuestra intención será modelar un nodo emisor y uno receptor, comunicados por canales no confiables, es decir por canales sujetos a la perdida de mensajes. Buscaremos verificar con nuestra herramienta si este protocolo es tolerante a un numero de ocurrencia finito de estas fallas sobre los canales, en cuanto a que eventualmente todos las tramas puedan ser entregadas al receptor.

Los algoritmos (1) y (4), y el modelado del emisor.

El siguiente pseudocódigo representa el algoritmo de envío de trama de secuencia:

```
if( buffer de envio vacio )
    if( estoy restransmitiendo )
        continuo con la retransmision;
    else if( tengo tramas y la ventana no esta llena )
        envio trama nueva;
    else
        inicio retransmision;
else
```

continuar

---

mientras que este otro pseudocódigo representa al de recepción de confirmación de recepción:

---

if(la confirmacion cae dentro de la ventana de emision)
    deslizo la ventana;
    if(estoy retransmitiendo)
        dejo de retransmitir;
else
    descarto la confirmacion;

Como vemos, iniciaremos la retransmisión de tramas solo cuando hayamos enviado todos los paquetes dentro de nuestra ventana. A su vez detendremos la retransmisión al recibir tramas de confirmación válidas, ya que en ese caso se avanza la ventana y nuevas tramas se hacen disponibles para la retransmisión. Notemos además que suponemos que siempre tenemos paquetes nuevos para insertar en nuestras tramas y enviarlos.

El modelado del emisor en Falluto2.0 presenta entonces el siguiente aspecto:

```
-- DEFINICIONES
DEFINE WZ := 2          -- Window size.
DEFINE MSEQ := 2        -- Max sequence number.
DEFINE MSP1 := MSEQ + 1 -- Max sequence number + 1.

PROCTYPE sender( ACK; send, read)

VAR
    DTX:  0..2 -- Data in output buffer
    NS:   0..2 -- Next sequence number
    LA:   0..2 -- Last Ack
    RT:   0..2 -- Retransmition index
    SENT: bool -- Output buffer empty?
```

```
INIT
    NS = 0 & LA = 0 & DTX = 0 & SENT & RT =  0

TRANS

    -- CONTINUAR CON LA RETRANSMISION SI SE HA EMPEZADO
    [contRetr]: RT != NS & SENT
                =>
                DTX' = RT, RT' = (RT+1)%MSP1, SENT' = FALSE;

    -- NUEVO FRAME
    [new]:  RT = NS & SENT &
            ((NS >= LA & NS-LA < WZ)|(NS < LA & MSP1-LA+NS < WZ))
            =>
            SENT' = FALSE, NS' = (NS+1)%MSP1,
            RT' = (RT+1)%MSP1, DTX' = NS;

    -- COMENZAR RETRANSMISION SI LA VENTANA ESTA LLENA
    [startRetr]:
        ((NS>=LA & NS-LA >= WZ)|(NS<LA & MSP1-LA+NS >= WZ)) &
        SENT & RT = NS
        =>
        SENT' = FALSE, DTX' = LA, RT' = LA;


    -- LEER ACK INVALIDO Y DESCARTARLO
    [read]: !(NS > LA & (ACK > LA | ACK <= NS)) &
            !(NS <= LA & ACK > LA & ACK <= NS);

    -- LEER ACK VALIDO Y ACEPTARLO
    [read]: (NS>LA & (ACK>LA | ACK<=NS)) |
            (NS <= LA & ACK > LA & ACK <= NS)
            =>
            LA' = ACK, RT' = NS;


    -- ENVIAR EL MENSAJE POR EL CANAL DE COMUNICACION
```

```
    [send]: !SENT => SENT' = TRUE;

ENDPROCTYPE
```

Hemos simulado un emisor sin fallas, el cual memoriza el valor de la última confirmación de recepción, el número de secuencia del próximo paquete a ser enviado y el índice de retransmisión. De esta manera la ventana de transmisión queda definida entre los índices dados por LA y NS, y en caso de retransmisión, el índide RT recorre esta ventana simulando enviar las tramas correspondientes.

La sección *TRANS* del emisor, se ha definido tomando en cuenta el sentido de las construcciones *IF-ELSEIF-ELSE*. Por lo tanto la guarda de cada transición definida se conforma, en parte, por la negación de las guardas de las transiciones correspondientes a los casos anteriores dentro de la construcción *IF-ELSEIF-ELSE*. Las tres primeras transiciones se corresponden al algoritmo para envío de tramas de secuencia, mientras que las transiciones 4 y 5 pertenecen al de recepción de tramas de confirmación de recepción.

Los algoritmos (2) y (3), y el modelado del receptor.

Miremos ahora el pseudocódigo correspondiente a los algoritmos de recepción de tramas de secuencia y envío de confirmación de recepción. Estos algoritmos serán incluídos en el modelado del receptor.

El siguiente pseudocódigo describe el algoritmo de recepción de tramas de secuencia:

```
if( es la trama que esperaba )
    aceptar trama y correr ventana;
    generar confirmacion de recepcion;
else
    descartar trama;
    re-generar confirmacion de recepcion;
```

El algoritmo para envío de tramas de confirmación de recepción es muy

simple ya que solo consta de enviar el número de secuencia que se espera recibir en la próxima trama. Ya que en el caso del protocolo GO-BACK-N, la ventana de recepción posee tamaño 1, el algoritmo consistirá en enviar el número de secuencia correspondiente a la nueva ventana obtenida luego de la recepción de una trama válida.

Miremos a continuación nuestro modelado del receptor en el lenguaje de Falluto2.0:

```
PROCTYPE receiver( NF; send, read)

VAR
        NR:0..2 -- Next request number
        NA:bool -- Need to send ack

INIT
        NR = 0 & NA = FALSE

TRANS

        -- DESCARTAR TRAMA INVALIDA
        [read]: NF != NR => NA' = TRUE;

        -- RECIBIR TRAMA VALIDA
        [read]: NF = NR => NR' = (NR+1)%(MSP1), NA' = TRUE;

        -- ENVIAR ACK DE SER NECESARIO
        [send]: NA => NA' = FALSE;

ENDPROCTYPE
```

Hemos modelado entonces un receptor sin fallas, el cual mantiene memoria el número de secuencia de la próxima trama que se espera recibir. En su sección *TRANS* podemos distinguir el modelado de los algoritmos (2) y (3). Las dos primeras transiciones forman parte de la recepción de tramas de secuencia, mientras que la última transición describe el envío de la una trama de confirmación de recepción.

A continuación modelaremos los canales de comunicación entre el emisor y el receptor. Estos canales estarán sujetos a fallas, en particular a fallas que implican la perdida de los mensajes que intentan atravesarlos. A continuación encontramos el modelo de los canales en el lenguaje de Falluto2.0:

```
PROCTYPE channel( IN; send, read )

VAR
    buff:  array 0..2 of 0..1 -- fisical channel
    data:  0..2               -- output data
    empty: bool               -- output buffer empty?

FAULT
    loose0: buff[0] > 0 => buff[0]' = buff[0]-1 is TRANSIENT
    loose1: buff[1] > 0 => buff[1]' = buff[1]-1 is TRANSIENT
    loose2: buff[2] > 0 => buff[2]' = buff[2]-1 is TRANSIENT

INIT
    buff[0] = 0 & buff[1] = 0 & buff[2] = 0 & empty & data = 0

TRANS
    -- ENVIAR UNA TRAMA POR EL CANAL
    [send]: IN = 0 & buff[0] < MAXCOP => buff[0]' = buff[0]+1;
    [send]: IN = 1 & buff[1] < MAXCOP => buff[1]' = buff[1]+1;
    [send]: IN = 2 & buff[2] < MAXCOP => buff[2]' = buff[2]+1;

    -- UNA TRAMA LLEGA AL FINAL DEL CANAL
    [arrive]: empty & buff[0] > 0
              =>
              data' = 0, empty' = FALSE, buff[0]' = buff[0]-1;
    [arrive]: empty & buff[1] > 0
              =>
              data' = 1, empty' = FALSE, buff[1]' = buff[1]-1;
    [arrive]: empty & buff[2] > 0
              =>
              data' = 2, empty' = FALSE, buff[2]' = buff[2]-1;
```

```
    -- LA TRAMA ES LEIDA POR EL RECEPTOR DEL CANAL
    [read]: !empty => empty' = TRUE;

ENDPROCTYPE
```

En el modelado de los canales tenemos que la variable *buff* en la posición i define la cantidad de mensajes con número de secuencia i que presentes en el canal. Por otro lado la variable *data* representa la trama que ha arribado a destino para ser leída por el receptor. A su vez la variable *empty* indica si el receptor ya ha leído el dato que le llegó o no. Podemos distinguir tres tipos de transiciones en este modelado: las de nombre *send* representan la introducción de un mensaje en el canal por parte del emisor del mismo; las de nombre *arrive* representan el arribo de la trama al extremo del canal donde será leído; y por último la transición de nombre *read* representa la lectura de la trama por parte del receptor del canal.

Hemos modelado las fallas de manera muy sencilla gracias a la lenguaje declarativo que otorga nuestra herramienta para la inyección de las mismas. De esta manera en una simple linea de la forma:

$$loosei: \ buff[i] \ > \ 0 \ => \ buff[i]' \ = \ buff[i] - 1 \ is \ TRANSIENT$$

hemos definido la falla que representa la perdida de una trama de número de secuencia i. Notemos que el nombre de la falla es $loosei$, mientras que la condición de habilitación representa la necesidad de que en efecto exista un mensaje en el canal para que el mismo se pierda ($buff[i] > 0$). Por otro lado declaramos el efecto de perdida reduciendo la cantidad de tramas de número de secuencia i en el canal usando la post-condición $buff[i]' = buff[i] - 1$, y declaramos que la falla es de tipo $TRANSIENT$ indicando que la misma puede ocurrir reiteradamente en cualquier momento durante la ejecución siempre que se cumpla su condición de habilitación.

Por último veamos como modelar el sincronizado de los canales con el emisor y el receptor mediante el pasaje de nombres de acción comunes en la etapa de instanciación. Instanciamos entonces un emisor, un receptor, y dos canales, uno desde el emisor al receptor y otro en dirección contraria. Hacemos esto de la siguiente manera:

```
INSTANCE emisor   = sender   (buffRE.data , send1, read1)
INSTANCE receptor = receiver (buffER.data , send2, read2)
INSTANCE buffER   = channel  (emisor.DTX  , send1, read2)
INSTANCE buffRE   = channel  (receptor.NR , send2, read1)
```

De esta manera hemos logrado que el emisor pueda leer los datos que llegan a través del canal *buffER* al pasar como parámetro la variable *buffRE.data*. De manera similar el receptor recibe los datos desde el canal *bufferRE* a través de la variable *bufferRE.data* pasada como parámetro. A su vez cada canal puede leer los datos de salida de su emisor correspondiente a partir del pasaje de los parámetros *emisor.DTX* y *receptor.NR*. La sincronización de las acciones *read* y *send* tanto del emisor y el receptor con sus canales correspondientes se logra pasando como parámetro nombres equivalentes para las acciones.

Nos interesa probar ciertas propiedades sobre este modelo. Una de ellas expresa que cierto número de secuencia llegue siempre a menudo al receptor. Usemos en particular el número máximo de secuencia y revisemos si se cumple que el receptor lo recibe siempre a menudo. De acuerdo al modelado que hemos realizado podemos identificar la llegada de una trama con número de secuencia máximo como la ocurrencia de una lectura del canal por parte del emisor, en momento en que la variable *data* del canal correspondiente contiene el valor *MSEQ*. Queremos entonces describir en lenguaje de Falluto una propiedad que exprese que *para toda ejecución suceda que siempre a menudo canalER.data = MSEQ & receptor lee el canal*. Encontramos que la sintaxis de Falluto2.0 ofrece una manera muy sencilla de describir la acción en la cual el receptor lee el canal, usando la construcción *just()*. Obtenemos entonces usando el lenguaje de Falluto2.0. la siguiente fórmula LTL para expresar la propiedad que hemos sugerido:

Propiedad(1):

```
LTLSPEC G F (canalER.data = MSEQ & just(read2))
```

Podemos constatar la veracidad de esta propiedad bajo escenarios en los que las fallas no ocurren usando la meta-propiedad *NORMAL_BEHAIVIOUR*, y bajo escenarios en los que la ocurrencia de fallas es de carácter finito usando la meta-propiedad *FINITELY_MANY_FAULTS*. Las propiedades resultantes

del uso de estas meta-propiedades tienen la siguiente forma:

Propiedad(2):

```
NORMAL_BEHAIVIOUR → G F (canalER.data = MSEQ & just(read2))
```

Propiedad(3):

```
FINITELY_MANY_FAULTS → G F (canalER.data = MSEQ & just(read2))
```

Como resultado de la verificación realizada por Falluto2.0 obtendremos que solo las propiedades (1) y (3) se cumplen, mientras que un contraejemplo sera presentado para refutar la propiedad (2).

7.3. Discusión

El primer caso de estudio presentado en este capítulo fue inspirado en la otros trabajos que poseen verificaciones similares. Pudimos notar la facilidad de inyectar fallas en el sistema. Pudimos observar también la facilidad para extraer el modelo de la especificación y describirlo de manera directa a partir de nuestro lenguaje de modelado.

El segundo caso de estudio por su parte, fue extraído de un caso de investigación real. La especificación original contiene un error en sus algoritmos. El modelado y posterior verificación usando nuestra herramienta logró sacar a luz este error al devolver una ejecución de contraejemplo para la cual no se cumple la propiedad (3):

```
FINITELY_MANY_FAULTS → G F (canalER.data = MSEQ & just(read2))
```

Luego de inspeccionar este contraejemplo se llegó a la conclusión de que situaciones como la que describimos a continuación estaban causando que el sistema no sea capaz de tolerar las fallas en el canal de conversación. Para describir la ejecución en la que el sistema no cumple con la propiedad deseada, tomaremos en cuenta un emisor con tamaño de ventana 1 y un número máximo de secuencia 3. Así también, entenderemos que el receptor no envía confirmación de recepción si la trama no contiene el número de secuencia esperado, tal cual lo especificaba el algoritmo original. La siguiente ejecución entonces representa un caso en el que la propiedad (3) no se cumple:

1. Emisor envía trama con número de secuencia 1.

2. Receptor recibe trama con número de secuencia 1, envía confirmación de recepción, y aumenta en 1 el valor de su ventana de recepción.

3. La confirmacíon de recepción se pierde.

4. *Comienza ciclo:* Emisor reenvía trama con número de secuencia 1, ya que no ha recibido confirmación.

5. Receptor descarta trama con número de secuencia 1, ya que espera número de secuencia 2. *Fin de ciclo*

La inspección de la ejecución de contraejemplo otorgada por Falluto2.0, permitió llegar a esta conclusión. A partir de esto, se buscó resolver el problema y de esta manera mejorar los mecanismos de tolerancia a fallas de nuestro sistema. La nueva especificación responde al modelado que se detalló en este capítulo, en donde el receptor enviá confirmación de recepción también cuando recibe tramas incorrectas, más allá de no aceptarlas. Otras soluciones podrían haberse implementado, pero como ya dijimos esto va más allá del interés de este trabajo. El output detallado de nuestra herramienta, junto con el modelado del *receptor* del sistema original defectuoso, pueden encontrarse en el apéndice D de este trabajo.

Capítulo 8

Conclusión

Apéndices

Apéndice A

Manual de Falluto2.0

Mostraremos en este apéndice los pasos a seguir para preparar el entorno operativo para Falluto2.0. Detallaremos además como utilizarlo para la verificación de nuestros sistemas tolerantes a fallas, describiendo cada una de las opciones y utilidades que la herramienta ofrece.

A.1. Instalación de Falluto2.0

Previo a poder utilizar Falluto2.0 en nuestra PC debemos preparar el entorno en nuestro sistema Linux. Para ello precisamos poseer instalado tanto NuSMV versión 2.5.3 o superior, como así también Python versión 2.6.5 o superior. Tanto NuSMV como Python deben ser accesible mediante la ruta de busqueda para ejecución 'PATH'.

A.2. Opciones y utilización de la herramienta

El primer paso para la verificación del sistema es poseer un archivo (por convención terminado en .fll) en el cual se encuentre la descripción del sistema junto con las opciones de verificación y las propiedades a verificar (Ver el manual de usuario de Falluto2.0 para mayores indicaciones). Dentro de la carpeta de instalación de Falluto2.0 encontramos el script ejecutable del programa llamado *Falluto2.0*. Ejecutamos este script con las siguientes opciones para realizar la verificación:

uso: Falluto2.0 [-h] [–version] [-s path] [-co] filename

argumentos posicionales:

filename	Ruta del archivo de input donde se encuentra la descripción del sistema.

argumentos opcionales:

-h, –help	muestra este mensaje de ayuda.
–version	muestra la versión de este programa y sale.
-s path, –s path, -save path	guardar la versión compilada a NuSMV de este sistema en el archivo dado por la ruta 'path'.
-co	output con color.

Apéndice B

Sintáxis formal de Falluto

A continuación presentamos la sintáxis formal de Falluto2.0 en términos de Parsing Expression Grammars (PEG) y Regular Expressions (RE). Estas producciones pasan por alto los espacios en blanco, tabulaciones y saltos de linea. Consideramos aquí terminales a las letras en itálico y a los símbolos y puntuaciones entre comillas.

B.1. Palabras reservadas de Falluto2.0

Las siguientes palabras son de caracter reservado, son usadas para fines específicos en la especificación del sistema, y no pueden ser usadas como identificadores de variables o nombres.

RESERVED ⟵ *in / CHECK_DEADLOCK / OPTIONS / ENDOPTIONS / SYSNAME / just / is / FAIRNESS / COMPASSION / U / V / S / T / xor / xnor / G / X / F / H / O / Z / Y / PROCTYPE / ENDPROCTYPE / INSTANCE / TRANS / INIT / VAR / FAULT / TRUE / FALSE / AG / AX / AF / EX / EF / EG / INST_WEAK_FAIR_DISABLE / FAULT_FAIR_DISABLE / in / FINITELY_MANY_FAULT / FINITELY_MANY_FAULTS / LTLSPEC / CTLSPEC / DEFINE / FAIRNESS / COMPASSION/ NORMAL_BAHAIVIOUR*

B.2. Algunas producciones simples

Identificadores pueden contener '.' para indicar pertenencia a un proceso específico (por ejemplo instancia.variable). Nombres en cambio no.

IDENT	⟵	! RESERVED [a-zA-Z_](“.”[a-zA-Z0-9_]+)?
NAME	⟵	! RESERVED [a-zA-Z_][a-zA-Z0..9_]*
INT	⟵	-? ([0] / [1-9][0-9]*)
BOOL	⟵	“TRUE” / “FALSE”
EVENT	⟵	“just(” IDENT “)”
NEXTREF	⟵	IDENT “ ’ ”
RANGE	⟵	INT “..” INT
BOOLEAN	⟵	“bool”
SET	⟵	“{” (IDENT / INT / BOOL) (“,” (IDENT / INT / BOOL))* “}”
INCLUSION	⟵	IDENT “in” (SET / RANGE)

B.3. Expresiones

Las expresiones describen fórmulas booleanas o enteras. A continuación encontramos su sintaxis:

EXPRESION	⟵	PROP
PROP	⟵	CONJ ((− > / < − >) PROP) ?
CONJ	⟵	COMP ((| / &) CONJ) ?
COMP	⟵	PROD ((<= / >= / > / < / ! = / =) CONJ) ?
PROD	⟵	SUM ((‘∗’ / ‘÷’ / ‘%’) PROD) ?
SUM	⟵	VALUE ((‘+’ / ‘−’) SUM) ?
VALUE	⟵	(‘(’ PROP ‘)’ / INCLUSION / NEXTREF / IDENT / INT / BOOL / EVENT / ! VALUE / - VALUE)
NEXTLIST	⟵	NEXTASSIGN (‘,’ NEXTASSIGN)*
NEXTASSIGN	⟵	NEXTREF (= EXPRESION / “in” (SET / RANGE))

B.4. Proctypes

Un sistema falluto se modela con un encabezado de opciones de configuración (opcional) y una serie de objetos pertenecientes a la lista – PROCTYPE, DEFINE, INSTANCE, SPEC, CONTRAINT –:

SYSTEM	⟵	OPTIONS ? (DEFINE / PROCTYPE / INSTANCE / SPEC / CONTRAINT) *

OPTIONS	⟵	“OPTIONS” (“SYSNAME” [a-z0-9A-Z_.]* / “CHECK_DEADLOCK” / “FAULT_FAIR_DISABLE” / “INST_WEAK_FAIR_DISABLE”)* “ENDOPTIONS”
DEFINE	⟵	“DEFINE” IDENT := EXPRESION
PROCTYPE	⟵	“PROCTYPE” IDENT “(” CTXVARS ? SYNCACTS ? “)” PROCTYPEBODY “ENDPROCTYPE”
CTXVARS	⟵	IDENT (“,” IDENT)*
SYNCACTS	⟵	“;” IDENT (“,” IDENT)*
PROCTYPEBODY	⟵	VAR ? FAULT ? INIT ? TRANS ?
VAR	⟵	“VAR” VARDECL*
VARDECL	⟵	IDENT “:” (BOOLEAN / SET / RANGE)
FAULT	⟵	“FAULT” FAULTDECL*
FAULTDECL	⟵	NAME “:” (EXPRESION ? => NEXTEXPR ?) ? “is” (BYZ / STOP / TRANSIENT)
BYZ	⟵	“BYZ” “(” IDENT (“,” IDENT)*, “)”
TRANSIENT	⟵	“TRANSIENT”
STOP	⟵	“STOP” (“(” IDENT, (“,” IDENT)* “)”) ?
INIT	⟵	“INIT” EXPRESION ?
TRANS	⟵	“TRANS” TRANSDECL*

TRANSDECL ⟵ “[” NAME? “]” “:” EXPRESION? (=> NEXTEXPR)?

B.5. Instanciación

Para especificar que un proceso forma parte del sistema, debemos instanciar el mismo a partir de un proctype previamente definido. La sintaxis de instanciación de procesos es la que sigue:

INSTANCE ⟵ “INSTANCE” NAME “=” NAME “(” INSTPARAMS “)”

INSTPARAMS ⟵ ((IDENT / INT / BOOL) (“,” (IDENT / INT / BOOL))*) ?

B.6. Especificación de propiedades

Usamos las siguientes reglas para especificar las propiedades a verificar sobre el sistema modelado. Encontramos que podemos definir propiedades tanto en lógica LTL como en CTL. Podemos además hacer uso de metepropiedades predefinidas para facilitar la verificación:

Para la elaboración de propiedades LTL y CTL, se ofrece la siguiente sintaxis:

SPEC ⟵ CTLSPEC / LTLSPEC / NORMALBEHAIVIOUR / FINMANYFAULTS / FINMANYFAULT

CTLSPEC ⟵ "CTLSPEC" CTLEXP

CTLEXP ⟵ CTLVALUE CTLBINOP CTLEXP / (*A* / *E*) '[' CTLEXP *U* CTLEXP ']' / CTLVALUE

CTLBINOP ⟵ & / | / *xor* / *xnor* / − > / < − >

CTLVALUE ⟵ CTLUNOP CTLEXP /'(' CTLEXP ')' / EXPRESION

CTLUNOP ⟵ *! / EG / EX / EF / AG / AX / AF*

LTLSPEC ⟵ "LTLSPEC" LTLEXP

LTLEXP ⟵ LTLBOP / LTLUOP

LTLBOP ⟵ LTLUOP LTLBINOPS LTLEXP

LTLUOP ⟵ LTLUNOPS* LTLVAL

LTLVAL ⟵ EXPRESION / '(' LTLEXP ')'

LTLUNOPS ⟵ *! / G / X / F / H / O / Z / Y*

LTLBINOPS ⟵ *U / V / S / T / xor / xnor /* | / & / < − > / − >

A continuación encontramos las reglas sintácticas para el uso de las meta-propiedades de Falluto2.0:

NORMALBEHAIVIOUR ⟵ "NORMAL_BEHAIVIOUR" "-¿" (CTLEXP / LTLEXP)

FINMANYFAULTS ⟵ "FINITELY_MANY_FAULTS" − > LTLEXP

FINMANYFAULT ⟵ “FINITELY_MANY_FAULT” ‘(’ IDENT (‘,’ IDENT)* ‘)’ − > LTLEXP

B.7. Restricciones y fairness

Las siguientes son reglas sintácticas para la elaboración de restricciones de fairness sobre la verificación del sistema. Específicamente podemos definir dos tipos de restricciones de fairness: inconditional fairness (*FAIRNESS*) y strong fairness (*COMPASSION*).

CONTRAINT ⟵ FAIRNESS / COMPASSION

FAIRNESS ⟵ “FAIRNESS” EXPRESION

COMPASSION ⟵ “COMPASSION” ‘(’ EXPRESION ‘,’ EXPRESION ‘)’

Apéndice C

Ejemplo paradigmático de compilación

Apéndice D

Protocolo de comunicación satelital defectuoso

A continuación encontramos el modelado del *nodo receptor* segun la especificacion defectuosa del protocolo de comunicación satelital introducido en la discusión del capitulo 7, junto con la ejecucion de contraejemplo otorgada como resultado de la verificacion a traves de Falluto2.0.

El modelado del receptor

```
PROCTYPE receiver( NF; send, read)

VAR
        NR:0..2 -- Next request number
        NA:bool -- Need to send ack

INIT
        NR = 0 & NA = FALSE

TRANS
        -- DESCARTAR TRAMA INVALIDA
        [read]: NF != NR;
        -- RECIBIR TRAMA VALIDA
        [read]: NF = NR => NR' = (NR+1)%(MSP1), NA' = TRUE;
```

```
        -- ENVIAR ACK DE SER NECESARIO
        [send]: NA => NA' = FALSE;

ENDPROCTYPE
```

Ejecucion de contraejemplo

```
|-| Specification FINITELY_MANY_FAULTS
    GF((just( read2 )) & (buffER.data = MSEQ)) is false
```

as demonstrated by the following execution sequence:

```
---> State: 0 <---
  buffRE buff[0] = 0
  buffRE buff[1] = 0
  buffRE buff[2] = 0
  buffRE buff[3] = 0
  buffRE data = 0
  buffRE empty = TRUE
  receptor NR = 0
  receptor NA = FALSE
  buffER buff[0] = 0
  buffER buff[1] = 0
  buffER buff[2] = 0
  buffER buff[3] = 0
  buffER data = 0
  buffER empty = TRUE
  emisor DTX = 0
  emisor NS = 0
  emisor LA = 0
  emisor RT = 0
  emisor SENT = TRUE

@ [action] emisor / new

---> State: 1 <---
  emisor NS = 1
  emisor RT = 1
  emisor SENT = FALSE

@ [Synchro] send1 [buffER/send || emisor/send]

---> State: 2 <---
  buffER buff[0] = 1
  emisor SENT = TRUE

@ [action] buffER / arrive

---> State: 3 <---
  buffER buff[0] = 0
```

```
  buffER empty = FALSE

@ [Synchro] read2 [buffER/read || receptor/read]

---> State: 4 <---
  receptor NR = 1
  receptor NA = TRUE
  buffER empty = TRUE

@ [action] emisor / new

---> State: 5 <---
  emisor DTX = 1
  emisor NS = 2
  emisor RT = 2
  emisor SENT = FALSE

@ [Synchro] send1 [buffER/send || emisor/send]

---> State: 6 <---
  buffER buff[1] = 1
  emisor SENT = TRUE

@ [action] buffER / arrive

---> State: 7 <---
  buffER buff[1] = 0
  buffER data = 1
  buffER empty = FALSE

@ [Synchro] read2 [buffER/read || receptor/read]

---> State: 8 <---
  receptor NR = 2
  buffER empty = TRUE

@ [Synchro] send2 [buffRE/send || receptor/send]

---> State: 9 <---
  buffRE buff[2] = 1
  receptor NA = FALSE

@ [fault] buffRE / loose2 / Transient

---> State: 10 <---
  buffRE buff[2] = 0

@ [action] emisor / startRetr

---> State: 11 <---
  emisor DTX = 0
  emisor RT = 0
  emisor SENT = FALSE

@ [Synchro] send1 [buffER/send || emisor/send]

---> State: 12 <---
  buffER buff[0] = 1
  emisor SENT = TRUE
```

```
@ [action] emisor / contRetr

---> State: 13 <---
  emisor RT = 1
  emisor SENT = FALSE

@ [action] buffER / arrive

---> State: 14 <---
  buffER buff[0] = 0
  buffER data = 0
  buffER empty = FALSE

@ [Synchro] send1 [buffER/send || emisor/send]

---> State: 15 <---
  buffER buff[0] = 1
  emisor SENT = TRUE

@ [action] emisor / contRetr

---> State: 16 <---
  emisor DTX = 1
  emisor RT = 2
  emisor SENT = FALSE

@ [Synchro] read2 [buffER/read || receptor/read]

---> State: 17 <---
  buffER empty = TRUE

@ [Synchro] send1 [buffER/send || emisor/send]

---> State: 18 <---
  buffER buff[1] = 1
  emisor SENT = TRUE


>> Loop starts here <<

@ [action] buffER / arrive

---> State: 19 <---
  buffER buff[1] = 0
  buffER data = 1
  buffER empty = FALSE

@ [Synchro] read2 [buffER/read || receptor/read]

---> State: 20 <---
  buffER empty = TRUE

@ [action] emisor / startRetr

---> State: 21 <---
  emisor DTX = 0
  emisor RT = 0
  emisor SENT = FALSE
```

```
@ [action] buffER / arrive

---> State: 22 <---
  buffER buff[0] = 0
  buffER data = 0
  buffER empty = FALSE

@ [Synchro] send1 [buffER/send || emisor/send]

---> State: 23 <---
  buffER buff[0] = 1
  emisor SENT = TRUE

@ [Synchro] read2 [buffER/read || receptor/read]

---> State: 24 <---
  buffER empty = TRUE

@ [action] emisor / contRetr

---> State: 25 <---
  emisor RT = 1
  emisor SENT = FALSE

@ [action] buffER / arrive

---> State: 26 <---
  buffER buff[0] = 0
  buffER empty = FALSE

@ [Synchro] send1 [buffER/send || emisor/send]

---> State: 27 <---
  buffER buff[0] = 1
  emisor SENT = TRUE

@ [Synchro] read2 [buffER/read || receptor/read]

---> State: 28 <---
  buffER empty = TRUE

@ [action] emisor / contRetr

---> State: 29 <---
  emisor DTX = 1
  emisor RT = 2
  emisor SENT = FALSE

@ [Synchro] send1 [buffER/send || emisor/send]

---> State: 30 <---
  buffER buff[1] = 1
  emisor SENT = TRUE

@ [action] buffER / arrive

---> State: 31 <---
  buffER buff[1] = 0
  buffER data = 1
  buffER empty = FALSE
```

Apéndice E

extra

- falencias de falluto??? - fallas por omisión de input. - recuperción de fallas por parte del usuario.

Bibliografía

[1] Clarke, J.A.; Pradhan, D.K. *Fault injection: a method for validating computer-system dependability.* June.1995

[2] Steiner-etal:DSN04; Wilfried Steiner and John Rushby and Maria Sorea and Holger Pfeifer; *Model Checking a Fault-Tolerant Startup Algorithm: From Design Exploration To Exhaustive Fault Simulation.* The International Conference on Dependable Systems and Networks, IEEE Computer Society, Florence, Italy, june, 2004

[3] Theo C. Ruys and Ed Brinksma. *Model Checking: Verification or Debugging?* Faculty of Computer Science, University of Twente. P.O. Box 217, 7500 AE Enschede, The Netherlands.

[4] Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri and Andrei Tchaltsev. *NuSMV 2.5 User Manual.* `http://nusmv.fbk.eu/NuSMV/userman/v25/nusmv.pdf`

[5] Edgardo E. Hames. FALLUTO: Un model checker para la verificación de sistemas tolerantes a falla. Trabajo de grado, FaMAF, Universidad Nacional de Córdoba, 2009.

[6] Nicolás Bordenabe. OFFBEAT: Una extensión de PRISM para el análisis de sistemas temporizados tolerantes a fallas. Trabajo de grado, FaMAF, Universidad Nacional de Córdoba, 2011.

[7] *Python.* `http://www.python.org/about/`

[8] Bryan Ford. *Parsing Expression Grammars: A Recognition-Based Syntactic Foundation.* Massachusetts Institute of Technology; Cambridge, MA.

[9] FELIX C. GRTNER. *Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments.* Darmstadt University of Technology. ACM Computing Surveys, Vol. 31, No. 1, March 1999

[10] Jeffrey A. Clarke Mitre Corporation, Dhiraj K. Pradhan. *Fault Injection. A Method For Validating Fomputer-System Dependability.* Texas A&M University. June 1995

[11] Flavin Cristian. *Understanding fault tolerant distributed systems.* COMMUNICATIONS OF THE ACM, February 1991, Vol.34, No.2

[12] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* The MIT Press Cambridge, Massachusetts London, England.

[13] Edmund M. Clarke, Orna Grumberg, David E. Long. *Model checking.* NATO ASI DPD 1996: 305-349

[14] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008.

[15] Michael R A Huth, Mark D Ryan. *Logic in Computer Science, Modeling and reasoning about systems.* Cambridge University Press. 2000.

[16] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri. NuSMV*: a new Symbolic Model Verifier.* In N. Halbwachs and D. Peled, editors. Proceeding of International Conference on Computer-Aided Verification (CAV'99). In Lecture Notes in Computer Science, number 1633, pages 495-499, Trento, Italy, July 1999. Springer.

[17] A. Arora and M. G. Gouda. *Closure and convergence: A foundation of fault-tolerant computing.* IEEE Trans. Software Eng., 19(11):10151027, 1993.

[18] P. F. Castro, C. Kilmurray, A. Acosta, N. Aguirre. *dCTL: A Branching Time Temporal Logic for Fault-Tolerant System Verification.* SEFM 2011.

[19] Cimatti, Alessandro; Clarke, Edmund M.; Giunchiglia, Enrico; Giunchiglia, Fausto; Pistore, Marco; Rovere, Marco; Sebastiani, Roberto; and Tacchella, Armando. *NuSMV 2: An OpenSource Tool for Symbolic Model Checking.* Computer Science Department. Paper 430. 2002. `http://repository.cmu.edu/compsci/430`