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

Raul Monti

Diciembre 2012

ACA VA EL RESUMEN

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 [perdidas] monetarias y hasta llegar a ocasionar la [perdida] 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 especificón plantea) 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 correctitud de los sistemas tolerantes a fallas. El diseño de algoritmos de tiempo real distribuidos tolerantes a fallas es notoriamente dificil y *propenso a errores*: la combianación 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 autimática 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 la 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 con el usuario de este método 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 logre contribuir a disminuir los problemas al momento de verificar sistemas tolerantes a fallas. Por un lado se intenta evitar la introducción de errores en el modelado del sistema en el que conviven fallas con procesos concurrentes. Por otro lado se busca evitar la introducción excesiva de nuevas variables al representar el comportamiento de las fallas, evitando así la nociva explosión de estados al momento de la verificación.

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 errors evitando que el usuario deba explicitar el funcionamiento de la falla dentro del modelo.
Este trabajo se presenta como extensión tanto del trabajo realizado por Edgardo Hammes[5] como del 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. ¡Normalmente la confiabilidad es definida estadísticamente, indicando la probabilidad de que el sistema sea funcional y provea el servicio esperado en un algún momento específico ¿? [9].

2.1. Fallas

Intuitivamente, y dentro del contexto que nos competen, podemos definir a una falla como una desviación de su función esperada en una componente de hardware o software. Estas fallas pueden ocurrir en cualquier etapa de la evolución del sistema computacional en cuestión -especificación, diseño, desarrollo, elaboración, ensamblado, e instalación- y durante toda 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].

2.2. Sistemas tolerantes a fallas

Existen sistemas diseñados para ser tolerantes a fallas: ellos o bien exhiben un bien definido comportamiento ante fallas cuando estas ocurren, 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 en los puntos particulares de se ejecución en los cuales estas fallas ocurren, sin tener efectos permanentes sobre el mismo. 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 **transient**.

2.3. Conclusión

En este trabajo entonces no realizaremos mayor diferencia formal entre transiciones buenas del sistema y transiciones debido a fallas, ambas serán consideradas como transiciones posibles del sistema que podrán afectar o no el estado del mismo.

Distinguiremos además como dijimos dos clases de falla, las de tipo *Permanente* y las de tipo *Transient*.

Por último, nos ocuparemos del modelado del sistema en general y nos centraremos en la inyección de las fallas en el mismo. No nos detendremos sin embargo en temas específicos al diseño de la tolerancia a las 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, para proveer de técnicas de verificación más afectivas y reducir el tiempo invertido en aplicarlas[12].

El **Model Checking** es un método formal para la verificación exhaustiva de sistemas finitos. Logra esta verificación explorando todos los estados posibles del sistema con la intención de verificar la veracidad de una propiedad 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 transición

3.1.1. Labelled transition systems - LTS

Llamamos sistema de transiciones etiquetadas a un concepto de maquina abstracta usada en parte para el modelado de sistemas computacionales con-

currentes. 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 denominados 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) pertenece a R, entonces estamos indicando que existe una transición con nombre l desde el estado s1 al estado s2.
Así podemos modelar sistemas computacionales tomando cada elemento en S como un estado particular del sistema y definiendo relaciones etiquetadas entre los mismos para representar el comportamiento del sistema.

En particular nos interesa para este trabajo la capacidad que nos 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 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 [interliving] 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'.

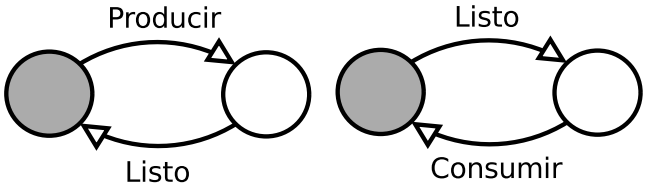

Figura 3.1: El productor (izquierda) y el consumidor (derecha).

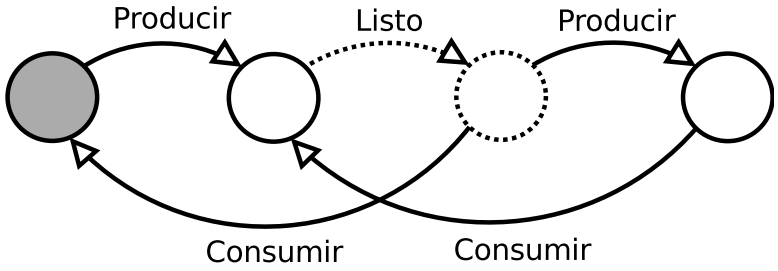

Figura 3.2: productor y consumidor de la figura 3.1 sincronizados.

3.1.2. Estructuras de Kripke

En model checking usamos un tipo de grafos de transición de estados llamados *Structuras de Kripke* para intuitivamente captar el comportamiento del sistema a modelar. 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 interesan a la verificación de sistemas vía model checking.

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 inciales.
3. R $\subseteq$ S x 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 R(s,s').
4. L : S x 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 $R(s_i, s_{i+1})$ vale para todo i ≥ 0.

Notemos que podemos traducir la representación LTS de un sistema a una representación en 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 su descripción $M_2 = (S_2, S_{2_0}, R_2, L_2)$ en términos de estructuras de Kripke sobre AP de la siguiente manera:

- $AP = \{action = e \mid e \in L_1 \cup \{null\}\}$
- $S_2 = S_1 \times (L_1 \cup \{null\})$
- $S_{2_0} = S_{1_0} \times \{null\}$
- $R_2 = \{(s, a) \rightarrow (s', b) \mid s \xrightarrow{b} s' \in R_1, a \in L \cup \{null\}\}$
- $L_2(s, a) = (action = a),\ \forall\ (s, a) \in S_2$

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 3.6. Notemos que el sistema traducido puede ser depurado quitando estados no alcanzables como se muestra en la tercera figura de 3.6.

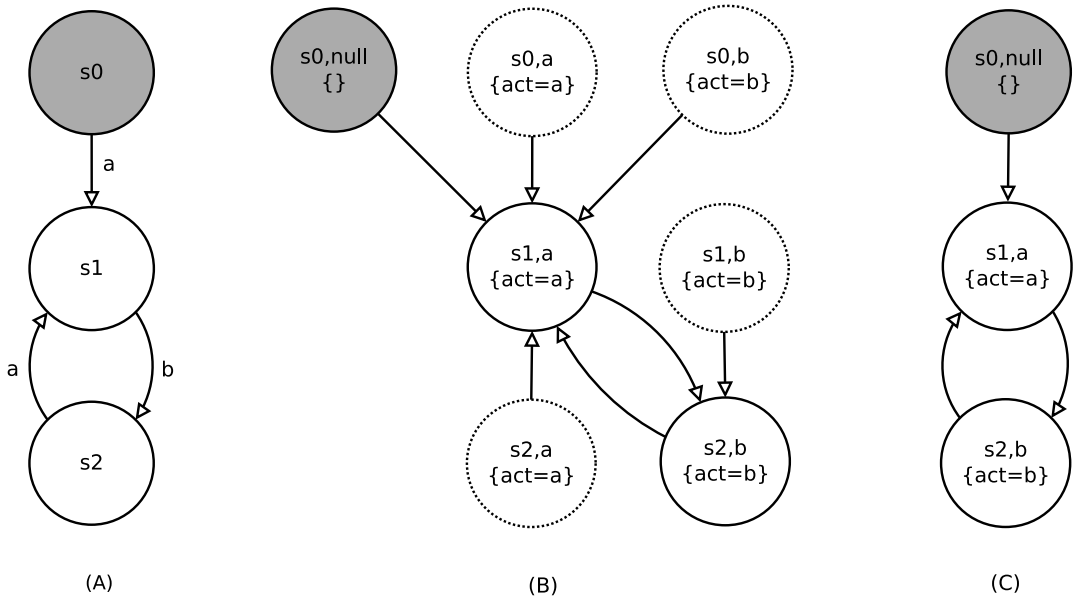

Figura 3.3: Ejemplo de traducción de un sistema LTS a estructuras de Kripke. (A) El sistema en LTS; (B) El sistema en estructuras de Kripke; (C) El sistema en estructuras de Kripke depurado

3.2. De estructuras de Kripke a fórmulas lógicas

Si bien las estructuras de Kripke nos 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 de primer orden. A continuación veremos como lograr la interpretación de estructuras de Kripke usando fórmulas lógicas de primer orden. En lo que a nosotros concierne, la lógica de primer orden estará comprendida por los conectivos lógicos usuales - $\neg, \wedge, \vee, \rightarrow, etc...$ -, y haremos uso también de los cuantificadores $\forall$ y $\exists$.

Supongamos que queremos modelar un sistema P, y tomemos todas sus variables de sistema $V = v_0, v_1, ..., v_n$. 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 valución estaría definiendo el estado del sistema. Por lo tanto en términos de estructuras de Kripke tenemos que si $M = (S, S_0, R, L)$ sobre AP modela un sistema con variables en V entonces:

- *Usualmente* $AP = \{v = d \mid v \in V, d \in D\}$.
- *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. Asi 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 es válida al valuar 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 funciones booleanas son un fuerte formalismo para representar los sistemas de transiciones y razonar acerca de propiedades sobre ellos. Se busca entonces poseer una eficiente representación 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 búsqueda binaria*. 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 formula 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'.

Si B es un BDD entonces 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'. Repito este paso 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 unos 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 subBDDs 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 formula 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' tal como sugiere R3, y finalmente D es el resultado de aplicar la regla R2 eliminando el nodo de etiqueta 'x'.

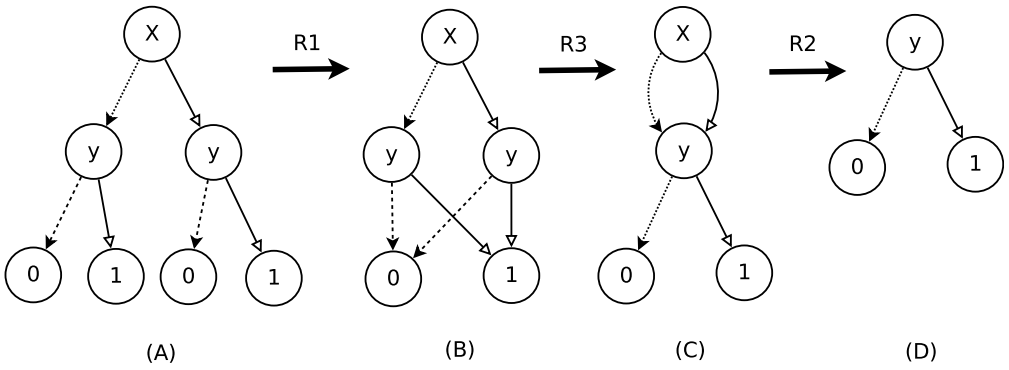

Figura 3.4: Reducción de un BDD según reglas R1,R2 y R3

Si bien se logra con estas reglas 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, pasa que* $i < j$*. 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 de equidad entre los mismos se reduzca a una simple comparación entre sus nodos. 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 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 como $f \wedge \neg g$. Si el OBDD canónico es distinto de B_0 entonces la implicancia 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

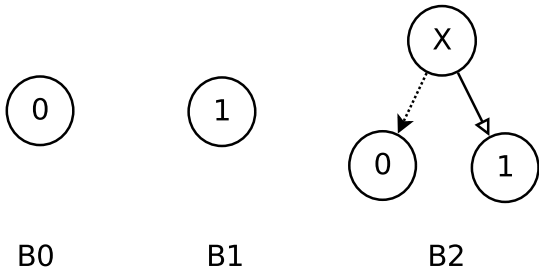

Figura 3.5: OBDDs canónicos de relevancia

que sugiere no representar cada estado por separado, sino que se representa en los OBDDs conjuntos de estados de manera simbólica, más precisamente a aquellos estados que cumplen con la propiedad que esta siendo verificada.Veamos ahora como representar estos conjuntos de estados en OBDDs y como representar la relación de transición entre ellos:

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. Tomaremos la primicia de que poseemos un orden sobre las proposiciones en AP dado como $(x_1, x_2, ..., x_n)$ y representaremos ahora cada estado s como un vector $(v_1, v_2, ..., v_3)$ donde cada $v_i \in 0, 1$ 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 representa 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 representa 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 x_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 de lo contrario, y $v'_i = 1$ si $p_i \in L(s')$ y 0 de lo contrario. En términos de OBDD, representamos la relación con el OBDD para 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 sistemas de reglas y símbolos que nos 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.

3.4.1. LTL

La **Lógica de Tiempo Lineal (LTL)** nos permite razonar sobre ejecuciones lineales a través del tiempo. En cada instante de tiempo solo existe una única ejecución real a realizarse. Comúnmente esta ejecución de la que hablamos comienza en este momento y se extiende al infinito. El tiempo es discreto y podemos realizar un paralelismo entre el salto de un momento al otro y cada paso en la ejecución del sistema que se analice. Así, cada momento 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 **G**lobally, **F**inally, Ne**X**t, **U**ntil y **R**elease.

A continuación damos una descripción intuitiva del significado de cada conectivo:

- G ϕ expresa que en todo momento vale ϕ.
- F ϕ expresa que ϕ se cumple en elgún momento.
- X ϕ expresa que ϕ se cumple en el momento inmediatamente posterior al actual.
- ϕ U ψ expresa que ψ vale en algún momento, y para todo momento anterior a aquel ϕ vale.
- ϕ R ψ expresa que o bien ϕ no vale nunca y ψ vale siempre, o bien ψ vale en cada momento hasta que ϕ valga.

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

Es común interpretar las fórmulas LTL sobre estructuras de Kripke. Una fórmula LTL puede ser satisfecha por una serie infinita de valuaciones sobre AP. Podemos ver a esta serie infinita como una palabra sobre una ejecución sobre una estructura de Kripke. De este modo si queremos saber si la formula ϕ se satisface en un sistema representado por la estructura de Kripke M, basta con ver que el lenguaje de M (todas las palabras posibles en M) satisfagan ϕ. Definimos la semántica de formulas 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 formula LTL con respecto a la palabra ω a partir de las siguientes reglas:

- $\omega \models p$ si $p \in \omega[0]$
- $\omega \models \neg p$ si no $\omega \models p$
- $\omega \models \phi \vee \psi$ si $\omega \models \phi$ or $\omega \models \psi$
- $\omega \models$ X ϕ si $\omega[1...] \models \phi$

- $\omega \models \phi U \psi$ si existe $i \geq 0$ tal que $\omega[i...] \models \psi$ y para todo $0 \leq k < i$, $\omega[k...] \models \phi$

Notar que los demás conectivos son derivados de aquellos para los que hemos definido las reglas de satisfactibilidad.

Muchas veces es de interés en model checking definir propiedades bajo **condiciones de equidad**, también llamadas *condiciones de* ***fairness***. 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: G F p (siempre finalmente se cumple p, o a menudo se cumple p)

2. Equidad fuerte: G F q $\rightarrow$ G F p (si a menudo vale q entonces a menudo vale p)

3. Equidad débil: F G q $\rightarrow$ G F p (si finalmente siempre vale q, entonces a menudo vale p)

3.4.2. CTL

A diferencia de LTL, en la **Lógica de Árbol Computacional (CTL)** no existe en cada momento un único camino a seguir. CTL es una lógica de tiempo ramificado, en cada momento consideramos todos los posibles saltos hacia un estado posterior, y por lo tanto consideramos diferentes ejecuciones como posibles a realizarse en el futuro. 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). A su vez establece la regla de estar obligado a usar estos cuantificadores delante de cada conectivo temporal, definiendo así fórmulas sobre estados y no sobre caminos como lo hacen la lógica LTL. CTL, a diferencia de LTL, nos permite hablar sobre la existencia de al menos un camino. Así es que podemos especificar propiedades como $E\ X\ p$ y $AG\ EF\ p$, 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 $\neg, \wedge, AX, AU, EU$ 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.
- $AX\ q$ se cumple en un estado s si para cualquier ejecución, q vale en el estado que sucede a s.
- $EX\ q$ se cumple en un estado s si existe al menos una ejecución donde q vale en el estado que sucede a s.
- $E[p\ U\ q]$ se cumple en un estado s si existe al menos una ejecución $s_1 s_2 ... s_n ...$ con $s_1 = s$ donde q vale en s_n y p vale para todo estado en $s_1 ... s_{n-1}$.

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 ∈ S y p ∈ AP.

- $M, s \models p \quad \leftrightarrow \quad p \in L(s)$
- $M, s \models \neg\phi \quad \leftrightarrow \quad$ no $M, s \models \phi$
- $M, s \models \phi_0 \wedge \phi_1 \quad \leftrightarrow \quad M, s \models \phi_0$ y $M, s \models \phi_1$
- $M, s \models AX\phi \quad \leftrightarrow \quad \forall (s, s') \in R,\ M, s' \models \phi$
- $M, s \models EX\phi \quad \leftrightarrow \quad \exists (s, s') \in R,\ M$ tal que $s' \models \phi$
- $M, s \models E(\phi_0 U \phi_1) \quad \leftrightarrow \quad \exists$ una ejecución $s_1 s_2 s_3 ... s_n ...$ definida por R tal que $s_0 = s, M, s_n \models \phi_1, \forall 0 < i < n\ M, s_i \models \phi_0$

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 raramente posible.
El término *verificación deductiva* normalmente refiere al uso de axiomas y reglas para probar la correctitud del sistema. Este método si bien posee la ventaja de poder probar correctitud 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.

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 model checking sufre del problema de explsión 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 preservar la no satisfactibilidad de las propiedades a verificar, en el caso que esta exista. Como vemos este proceso de abstracción y refinamiento requiere muchas veces de personal experto, y es una de las barreras a superar si se desea lograr que el model checking se convierta enteramente en una

herramienta “push-button”.

Christel Baier y Joost-Pieter Katoen en su libro ”Principles of model checking”[14] identifican las siguientes tareas como pasos para realizar el *proceso de model checking* sobre el diseño de un sistema:

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.

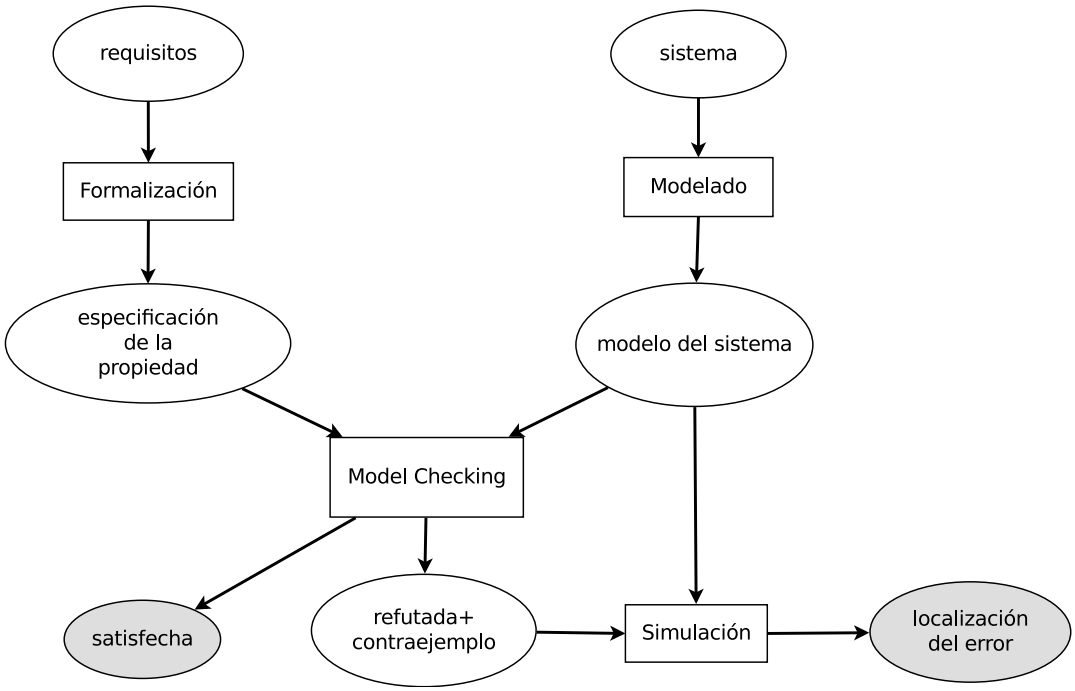

Figura 3.6: El proceso de Model Checking

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 checkin, 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 tal caso se deberá recurrir al refinamiento del sistema para conseguir disminuir la explosión de estados, o bien se deberá usar un equipo de mayor potencial en el caso de que ya no se logre llegar a un refinamiento significativo. Puede ser el caso de 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 usualmente un contraejemplo otorgado por la herramienta 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 correctitud 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.

Capítulo 4

NuSMV

4.1. El model checker NuSMV

NuSMV es un model chcker 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[**?**].

NuSMV nos permite la verificación de propiedades en los formalismos LTL y CTL, sobre la especificación de sistemas finitos [síncronos y asíncronos], bajo condiciones de fairness.

NuSMV2, el cual es utilizado en este trabajo, 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 nos presenta 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 Falluto 2.0.

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 primero en ser [...]. 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 4 booleanos indexado desde el -1 al 2. Recordemos 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.

ESTADO INICIAL

Para definir el conjunto de estados inicial de cada proceso introducimos una nueva sección dentro del módulo usando la palabra clave **INIT**, y hacemos 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 **??** 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 nos 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 formula 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 nos permite establecer restricciones de fairness sobre la ejecución de nuestro sistema. Esto nos 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 $G\ F\ q$.

- **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 $G\ F\ p \rightarrow G\ F\ q$.

PROPIEDADES Y VERIFICACIóN

NuSMV nos 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
```

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 sobre el sistema. Para una referencia aún más completa y precisa sobre las reglas sintácticas, en el apéndice **??** de este mismo trabajo se puede encontrar una descripción de la sintaxis en términos de Parsing Expresion Grammars. Podemos encontrar ejemplos esclarecedores también en ref.

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* comprende en su totalidad el funcionamiento de una clase de proceso, y los procesos reales pueden ser instanciados a partir del mismo. La declaración de un *proctype* es introducida mediante la palabra clave *PROCTYPE*. A continuación se establece un nombre para esta clase de procesos, y se especifica si poseerá o no variables de contexto, y acciones de sincronización. Podemos decir que todo esto forma parte del encabezado del *proctype*:

```
PROCTYPE miProceso ( ctxVar1, ctxVar2 ; sinchroAct1, 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. Podemos entonces declarar variables de tipo:

   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 al ejemplo que sigue:

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

   d) **Vector.** Podemos declarar variables de tipo vector, eligiendo alguno de los anteriores como subtipo. El subtipo es el tipo de cada valor perteneciente al vector. De este modo podemos declarar vectores de booleanos, o vectores de enteros o de tipo enumerado. Se [contribuye] a la declaración de los vectores con un rango indicando la indexación del vector. El ejemplo que sigue es un vector de subtipo enumerado, y se indexa desde el -1 al 2:

   ```
   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 formula 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 mas 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 podernos referir a la misma en el momento de la verificación. Llamaremos acción al nombre de la transición. La sintaxis general para la declaración de una transición es la que sigue:

   ```
   [accion] condici\'on_de_habilitaci\'on => postcondici\'on ;
   ```

   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 *condición de habilitación* es una fórmula booleana sobre el estado actual del sistema, que restringe los estados de partida de la transición a aquellos que la cumplan. La postcondición 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 acción. 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;
   ```

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(parametro1, parametro2, ..., parametroN
```

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, usamos la construcción *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.

5.2. Lenguaje de descripción de fallas

En Falluto2.0 podemos inyectar fácilmente fallas en los procesos definidos. 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 inyección 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, afectando así el estado del 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 formulas puede encontrarse en el apéndice **??**.
Falluto2.0 posee además las siguientes meta-propiedades para la verificación del sistema:

- **Normal Behaiviour** Usamos esta 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 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 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 sobres 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 formulas 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.

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. Para ello es necesario tomar en cuenta la semántica de Falluto2.0. 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 la definición del comportamiento de los procesos, llevado a cabo a partir de los proctypes, de la instanciación de los diferentes procesos. Se busca que los procesos se desarrollen de manera concurrente en el entorno del sistema global, donde cada procesos 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 interliving 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 rellena 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. En la sección VAR entonces colocaremos por cada variables de cada instancia del sistema 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. De esta manera podremos hablar sobre propiedades a cerca del accionar del mismo y establecer restricciones sobre el mismo. El dominio de esta variables entonces 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 las 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 formulas booleanas 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 diseado 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 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. Nuestro módulo de NuSMV quedará 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 formulas 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 formula 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 **??**):

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

La versión compilada de esta acción tendrá el siguiente aspecto:

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

Notar que de darse esta acción, el nombre de la misma quedará guardado como valor de actvar en el próximo estado, indicando que la acción que se llevo a cabo para llegar a ese estado fue en efecto esta. *!faultact* es un

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 ende serán compiladas muy parecidas 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 formula booleana:

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

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

Como en el caso de las transiciones comunes, el valor de la variable de acción denotará haber llegado al estado siguiente a partir de la ocurrencia de esta 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 comunes.

TRANSICIONES DE EFECTOS DE FALLA BIZANTINA 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. *next(actvar)=bizefect* se encarga de colocar el valor correcto a la variable *actvar* con el fin de indicar cuál ha sido la acción que se ha ocurrido para llegar al nuevo estado modificado. Por último *permanecen* se forma por la conjunción de las formulas *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 destinos 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 restricciones de fairness. Notar que a falta de ella y en presencia de solo transiciones de falla habilitadas, estaríamos obligando la ocurrencia de una de estas lo cual no es correcto. La compilación de la transición de deadlock tiene entonces 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 solo realizamos la transición de deadlock cuando nos vemos inhabilitados 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.

6.3. Fairness de fallas y procesos

En general nos interesa trabajar sobre diseos 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. Esto 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 siempre a menudo se realice una transición buena de ser posible, de lo contrario se caiga en deadlock como de costumbre. Recordemos que para poder caer en deadlock, todas las transiciones normales deben estar deshabilitadas.

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 buena. Decimos entonces que esta es una **condición de fairness débil para procesos**. Para implementar esta condición pediremos entonces para 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 buenas están deshabilitadas. Agregamos entonces por defecto la siguiente condición de fairness a nuestro sistema compilado:

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

donde *proc_deadlock* es la formula que expresa que este proceso cayó en deadlock y la construimos como sigue:

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

donde $T_{N_{proc}}$ es el conjunto de transiciones normales del proceso y $habilitacion_t$ 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 este esta en deadlock. Por lo tanto decimos que la condición de fairness es débil y solo se asegurará atender al proceso si el mismo nunca cae en deadlock.

6.4. Semántica de propiedades

Como vimos al finalizar la sección **??** del capítulo **??**, Falluto2.0 nos 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

$$just(accion)$$

. Al momento de verificar, esta construcción nos permite expresar la ocurrencia de *accion* como acción para llegar al estado actual. Por lo tanto

just(accion1) es válido en un estado cualquiera solo si la acción realizada para llegar a ese estado fue *accion1*. Notemos que esta formula nunca vale 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. Por lo tanto la compilación se reduce a una simple traducción semejante a la que sigue: *just(accion1)* se traduce a *actvar = accion1*.

Vimos también que podemos definir propiedades en LTL y CTL usando la sintaxis $LTLSPEC\ q$ y $CTLSPEC\ p$ donde q es una formula LTL y p una CTL. La compilación de estas fórmulas se hace de manera directa, tomando en cuenta lo expresado en el párrafo anterior. Esto vale también para las condiciones de fairness definidas por el usuario con las sintaxis $FAIRNESS\ q$ y $COMPASSION(p, q)$.

Miremos entonces la semántica y compilación de las meta-propiedades que Falluto2.0 nos ofrece:

- **Meta-propiedad Finitely Many Fault/s.** Esta meta-propiedad nos 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 $FINMANYFAULT(f_1, f_2, ..., f_n) \rightarrow q$ donde $f_1, f_2, ..., f_n$ 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:

$$LTLSPEC\ (\ F\ !\ (\ actvar\ in\ \{f_1, f_2, ..., f_n\}\ )\ )\ \rightarrow\ q_{compilada}$$

- **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 formula LTL o una CTL. La compilación para el caso en que q esta redactada en LTL es la que sigue:

$$LTLSPEC\ (\ G\ !\ (\ actvar\ in\ \{f_1, f_2, ..., f_n\}\ )\ )\ \rightarrow\ q_{compilada}$$

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

$$CTLSPEC\ (\ AG\ !\ (\ actvar\ in\ \{f_1, f_2, ..., f_n\}\ )\ )\ \rightarrow\ q_{compilada}$$

En ambos casos $\{f_1, f_2, ..., f_n\}$ es el conjunto de nombres de todas las fallas declaradas en el sistema por el usuario.

Falluto2.0 nos ofrece la posibilidad de verificar que nuestro sistema no caiga en estado de **deadlock**. La instrucción definida por la sintaxis *CHECK-DEADLOCK* 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.

Capítulo 7

Casos de estudio

La intencin de este captulo es la de hacer un repaso sobre el uso de Falluto2.0 y sus funcionalidades. Para ello se mostrar el trabajo sobre una serie de problemas de carcter paradigmtico en el estudio de la verificacin 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 seccin razonando sobre los resultados expuestos por nuestra herramienta.

7.1. Commit atómico de 2 fases (2PC)

El commit atmico es un problema muy conocido en el mbito de las bases de datos y sistemas de revisin distribuidos entre otros. En estos sistemas es usual encontrar situaciones donde se requiere realizar operaciones de manera atmica, es decir en que estas operaciones se realicen de una sola vez, como si se tratase de una nica operacin. Muchas veces esto requiere de la sincronizacin y votacin de los distintos procesos que componen el sistema.

El protocolo de commit de dos fases (2PC) es un tipo de protocolo de commit atmico (ACP). Es un algoritmo distribuido el cual coordina a todos los procesos participantes en una transaccin atmica distribuida por la cual debern decidir si hacer commit de la transaccin 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 estar encargado de iniciar la propuesta de commit, y mas adelante tomar una decisin a partir de la votacin realizada por el resto de los nodos. Los dems nodos sern denominados *votantes*, estos comunmente poseen la informacin necesaria para realizar la transaccin, y sern los que voten por **SI** o por **NO** a la propuesta del *coordinador*.

La especificacin del algoritmo busca lo siguiente. Cada proceso votar por **SI** o por **NO**, llegando as a la decisin 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 decisin de commit.

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

3. Todos los procesos que llegan a una decisin, 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 decisin basada sobre los votos obtenidos, e informar del resultado de su decisin a cada uno de los procesos involucrados en la transaccin[].

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

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

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

- **j.p** Indica la fase del proceso. Comenzar con el valor 0, pasar a tener valor 1 cuando el proceso haya emitido su voto, y finalmente valdr 2 cuando el proceso haya tomado su decisin final.
- **j.d** Dependiendo de la fase en la que se encuentre el proceso esta variable representar el voto emitido (fase 1), o la decisin tomada (fase 2). El valor *TRUE* indicara voto a favor o decisin de commit, el valor *FALSE* indicar voto en contra, o decisin de abortar.
- **j.up** Esta variable indicar si el proceso est en ejecucin (tomando valor TRUE) o se ha detenido por algn 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:

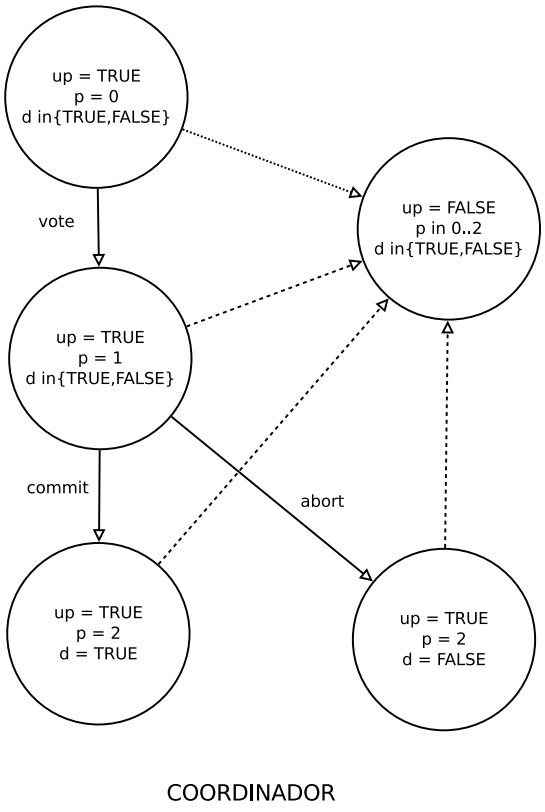

Figura 7.1: Diagrama de estados de una componente coordinador del protocolo de commit atmico de 2 fases

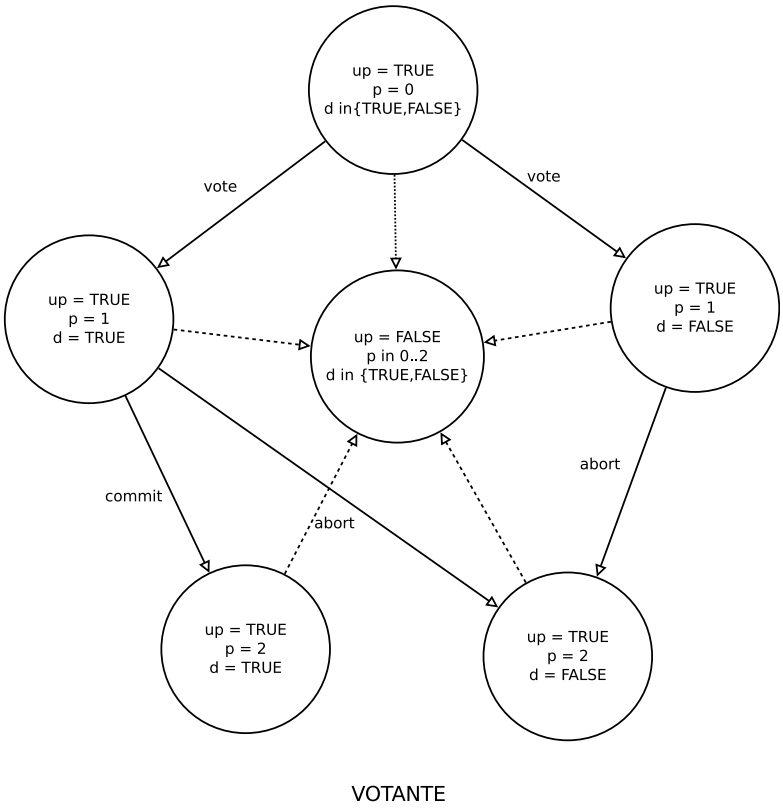

Figura 7.2: Diagrama de estados de una componente votante del protocolo de commit atmico de 2 fases

En nuestro caso de estudio no tomaremos en cuenta fallas en la comunicacin entre los procesos, solo simularemos fallas de tipo STOP, simulando as la cada en inactividad de algn proceso interviniente. Quedar abstrada tambin aquellas fallas que llevan a que los procesos voten por no hacer commit ya que no conciernen al protocolo. Miremos entonces el modelado de este protocolo en Falluto2.0, tomando el caso de 4 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 especificacin. Miremos sin embargo con detencin la seccin **FAULT** del modelado del proceso coordinador. En ella encontramos la especificacin de una nica falla, representando la detencin del proceso. Esta falla es por ende de tipo *STOP*, y como vimos es de carcter permanente, es decir que una vez que ocurre, el

proceso instanciado quedar infinitamente inhabilitado para realizar cualquier transicin buena. Dado que es de nuestro inters que los dems procesos puedan enterarse de la ocurrencia de la falla, o mas bien de la detencin del proceso coordinador, hemos agregado a la falla de detencin la postcondicin *up' = FALSE*. Veremos mas adelante como los dems procesos pueden acceder a esta variable, en funcin de variable de contexto, y de esta manera enterarse de la situacin del coordinador. Vale la pena tambin revisar los parmetros de contexto de esta declaracin. Como vemos posee 4 variables de contexto: – *v0, v1, v2, v3*–. Cada una de estas ser una referencia a cada votante en el sistema, y sern especificados en la instanciacin al realizar el pasaje de parmetros. De este modo el proceso coordinador podr en cualquier momento acceder a las variables de estado de los votantes en condicin 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 especificacin del algoritmo. Vemos como cada proceso tiene acceso en forma de solo lectura a cada uno de los dems procesos en el sistema, incluyendo al coordinador (*coo*). La falla declarada es idntica a la declarada en el modelado del coordinador. Por ltimo la accin *commit* fue separada para cada caso en que se detecta la toma de decisin de un proceso diferente. Si bien comparten el nombre, solo una de ellas ser elegida para realizar la transicin.

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

```
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 instanciacin de cada proceso como parmetros de los dems procesos en el orden correspondiente de manera que tenga sentido el modelado de los proctypes. Esto se ve en el caso de los procesos votantes, donde el ndice del parmetro para el proceso coordinador en la instanciacin coincide con el ndice de la variable de contexto *coo* en el proctype *RegularVoter*.

Capítulo 8

Conclusión

Apéndices

Apéndice A

Manual de Falluto2.0

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

A.1. Instalacin 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 versin 2.5.3 o superior, como as tambin Python versin 2.6.5 o superior. Tanto NuSMV como Python deben ser accesible mediante la ruta de busqueda para ejecucin 'PATH'.

A.2. Opciones y utilizacin de la herramienta

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

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

argumentos posicionales:

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

argumentos opcionales:

-h, –help	muestra este mensaje de ayuda.
–version	muestra la versin de este programa y sale.
-s path, –s path, -save path	guardar la versin 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 especficos en la especificacin 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 especfico (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 formulas booleanas o enteras. A continuacin 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 / SPECIFICATION / 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 instanciacin 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 lgica LTL como en CTL. Podemos adems hacer uso de mete-propiedades predefinidas para facilitar la verificacin:

Para la elaboracin 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 continuacin encontramos las reglas sintcticas 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

CONTRAINT ⟵ FAIRNESS / COMPASSION

FAIRNESS ⟵ *FAIRNESS* EXPRESION

COMPASSION ⟵ *COMPASSION* '(' EXPRESION ',' EXPRESION ')'

Apéndice C

Ejemplo paradigmático de compilación.

Apéndice D

extra

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

Bibliografía

[1] Fault injection: a method for validating computer-system dependability Clarke, J.A.; Pradhan, D.K. 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] Model Checking: Verification or Debugging?; Theo C. Ruys and Ed Brinksma; Faculty of Computer Science, University of Twente. P.O. Box 217, 7500 AE Enschede, The Netherlands.

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

[5] Falluto: Un model checker para la verificación de sistemas tolerantes a fallas; Edgardo E. Hames; Facultad de Matemática, Astronomía y Física, Universidad Nacional de Córdoba; Córdoba, 14 de diciembre de 2009.

[6] Offbeat: Una extensión de PRISM para el análisis de sistemas temporizados tolerantes a fallas; Nicolás Emilio Bordenabe; Facultad de Matemática, Astronomía y Física, Universidad Nacional de Córdoba; 28 de Marzo de 2011

[7] Python; http://en.wikipedia.org/wiki/Python_(programming_language)

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

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

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

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

[12] Principles of model checking. Christel Baier and Joost-Pieter Katoen. 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. 2008. Principles of Model Checking (Representation and Mind Series). The MIT Press.

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