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 mas 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 mas 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 mas 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'.

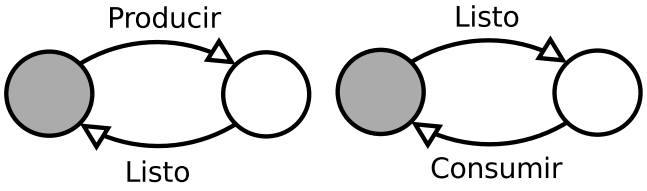

Figura 3.1: El productor (izquierda) y el consumidor (derecha).

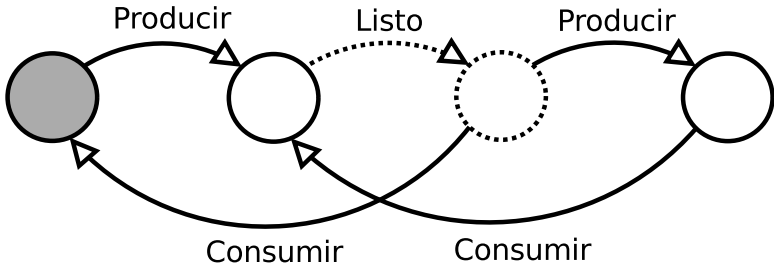

Figura 3.2: productor y consumidor de la figura 3.1 sincronizados.

3.1.2. Estructuras de Kripke

En model checking usamos un tipo de grafos de transicin 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 funcin que etiqueta cada estado con un conjunto de propiedades que son verdaderas en este estado. Los caminos en estas estructuras modelan la ejecucin del sistema[13]. Estas estructuras son lo suficientemente expresivas como para captar aspectos de lgicas temporales tales como (LTL y CTL) que interesan a la verificacin de sistemas va 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 atmicas 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 representacin 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 3.3. Notemos que el sistema traducido puede ser depurado quitando estados no alcanzables como se muestra en la tercera figura de 3.3.

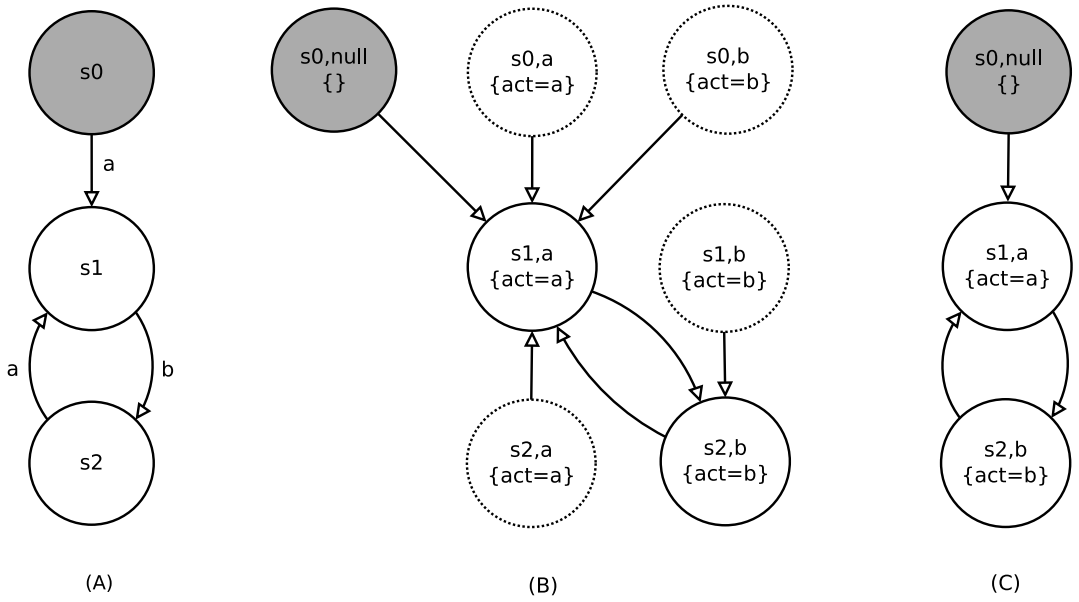

Figura 3.3: Ejemplo de traducción de un sistema LTS a estructuras de Kripke. (A) El sistema en LTS; (B) El sistema en estructuras de Kripke; (C) El sistema en estructuras de Kripke depurado

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 fórmulas lógicas en BDD

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. Comnmente 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 comn 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. As 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 diseo, 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"[**?**] 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 mas 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, diseo o propiedad.
     c) repetir todo el proceso.
   - La computadora se quedó sin memoria $\longrightarrow$ intentar reducir el modelo y empezar de nuevo.

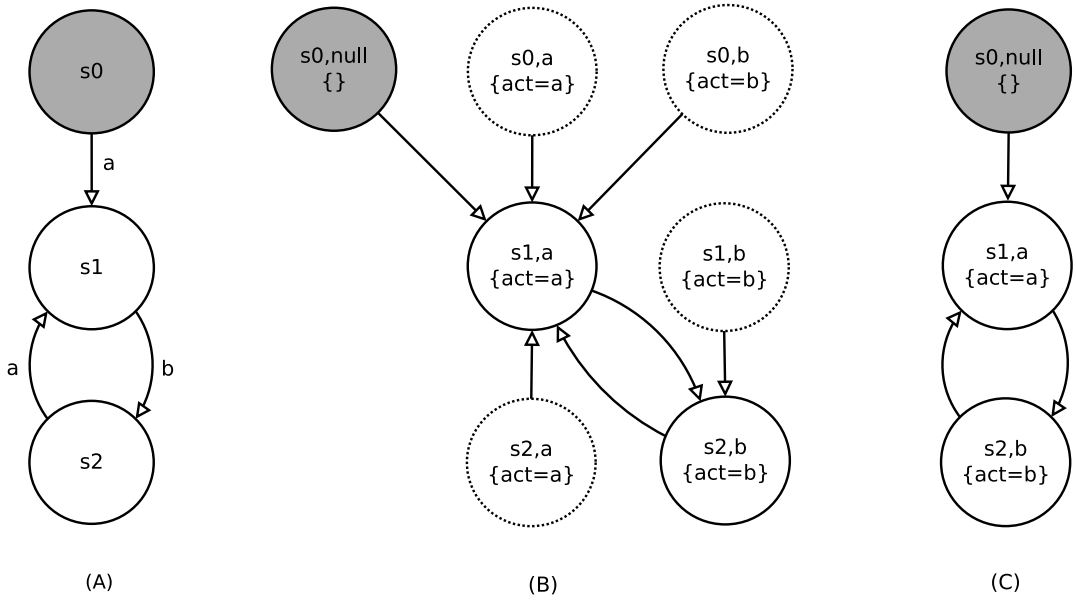
s0
a
s1
a
b
s2
(A)
s0,null
{}
s0,a
{act=a}
s0,b
{act=b}
s1,a
{act=a}
s1,b
{act=b}
s2,a
{act=a}
s2,b
{act=b}
(B)
s0,null
{}
s1,a
{act=a}
s2,b
{act=b}
(C)

Figura 3.4:

Capítulo 4

Casos de estudio

4.1. Commit atómico

4.2. Ejércitos bizantinos ?

4.3. Filósofos comensales ?

4.4. Falla bizantina ?

4.5. Elección de lider ?

4.6. Protocolos lamport

4.7. Feedback de contacto de Naza

4.8. Red satelital ultra secreta?

Capítulo 5

Conclusión

Capítulo 6

Apéndice A - Manual de Falluto2.0

Capítulo 7

Apéndice B - Sintáxis formal de Falluto2.0

Capítulo 8

Apéndice C - Ejemplo paradigmático de compilación.

Capítulo 9

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