

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

1. Introducción	7
2. Tolerancia a fallas	11
2.1. Fallas	11
2.2. Sistemas tolerantes a fallas	12
2.3. Conclusión	12
3. Model Checking	15
3.1. Sistemas de transición	15
3.1.1. Labelled transition systems - LTS	15
3.1.2. Estructuras de Kripke	16
3.2. Características del Model Checking	17
4. Casos de estudio	19
4.1. Commit atómico	19
4.2. Ejércitos bizantinos ?	19
4.3. Filósofos comensales ?	19
4.4. Falla bizantina ?	19
4.5. Elección de líder ?	19
4.6. Protocolos lamport	19
4.7. Feedback de contacto de Naza	19
4.8. Red satelital ultra secreta?	19
5. Conclusión	21
6. Apéndice A - Manual de Falluto2.0	23
7. Apéndice B - Sintáxis formal de Falluto2.0	25

8. Apéndice C - Ejemplo paradigmático de compilación.	27
9. extra	29

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 una falla funcional del sistema, provocando que se comporte de manera incorrecta, o simplemente deje de funcionar. Es importante entonces, para lograr una mayor confiabilidad del software (confiabilidad de que se comporte como su especificación 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 difícil y *propenso a errores*: la combinación de la ocurrencia de fallas, conviviendo con eventos concurrentes, y las variaciones en las duraciones de tiempos reales llevan a una explosión de estados que [genera una gran demanda] a la capacidad intelectual del diseñador humano[2].

En un mundo idealizado, los algoritmos son derivados por un proceso sistemático conducido por argumentos formales que aseguran su corrección respecto a la especificación original. En cambio, en la realidad contemporánea, los diseñadores suelen tener un argumento informal en mente y desarrollan el algoritmo final y sus parámetros explorando variaciones locales contra este argumento y contra escenarios que resalten casos difíciles o problemáticos. La exploración contra escenarios puede ser parcialmente automatizada usando un simulador y prototipos ágiles y esta automatización puede llegar a incrementar el número de escenarios que serán examinados y la confiabilidad de la examinación.

La examinación automática de escenarios puede ser llevada a un nivel aún más avanzado usando *Model Checking* [2]

En ciencias de la computación *Model Checking* refiere al siguiente problema: dada una estructura formal del modelo de un sistema, y dada una propiedad escrita en alguna lógica específica, verificar de manera automática y exhaustiva si el sistema satisface la propiedad. El sistema normalmente representa a un componente de hardware o software, y la fórmula a cumplirse representa una propiedad de *safety* o *liveness* que se desea verificar que el sistema cumpla, y de esta manera incrementar la confiabilidad sobre el mismo. El reducido nivel de interacción con el usuario de este método es visto como una ventaja para la aplicación en la industria, ya que incrementa la posibilidad de ser usado 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 errores 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 prerequisite 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 su 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 concurrentes. Este sistema de representación está compuesto por un conjunto de estados, un conjunto de etiquetas o nombres, y una relación ternaria expli-

cando las transiciones etiquetadas desde un estado hacia otro del sistema. Formalmente podemos decir que un LTS es una tres-upla $M = (S, L, R)$ donde:

- S es un conjunto (de estados)
- 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 s_1 y s_2 son elementos en S , l es un nombre en L , y (s_1, l, s_2) pertenece a R , entonces estamos indicando que existe una transición con nombre l desde el estado s_1 al estado s_2 .

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 la figura [□](#) podemos ver dos componentes ambas con su representación LTS. Podemos a partir de ellas definir un sistema concurrente sincronizado en el cual las transiciones con de igual nombre en cada componente deben ser ejecutadas de manera sincronizada. Vemos el sistema resultante en la figura [□](#).

3.1.2. Estructuras de Kripke

En model checking usamos un tipo de grafos de transición de estados llamados *Estructuras 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 via 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 iniciales.
3. $R \subseteq S \times S$ es una relación de transición que debe ser total, es decir, para cada estado $s \in S$ hay un estado $s' \in S$ tal que $R(s, s')$.
4. $L : S \times 2^{AP}$ es una función que etiqueta cada estado con el conjunto de proposiciones atómicas verdaderas en ese estado.

Una *[traza/camino]* en la estructura M desde el estado s es una secuencia infinita $\pi = s_0 s_1 s_2 \dots$ tal que $s_0 = s$ y $R(s_i, s_{i+1})$ vale para todo $i \geq 0$.

Notemos que podemos traducir una maquina abstracta en LTS a una en Estructuras de Kripke de manera muy sencilla. Sea $M_1 = (S_1, R_1, L_1)$ un sistema descrito en LTS, entonces construimos su descripción $M_2 = (S_2, S_0, R_2, L_2)$ en terminos de estructuras de Kripke sobre AP de la siguiente manera:

- Por cada e_i etiqueta en L_1 agregamos a AP la proposición $accin = e_i$
- Por cada tres-upla (s, l, s') en R_1 , agregamos la tupla (s, s')

3.2. 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 sobre 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:

- 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.

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 líder ?
- 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. - recuperación de fallas por parte del usuario.

Bibliografía

- [1] Fault injection: a method for validating computer-system dependability
Clark, 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 Interna-
tional Conference on Dependable Systems and Networks, IEEE Com-
puter 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 tem-
porizados 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_\(programming_language\)](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. Clark 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