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 actividades, 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 ocaciona daños mayores. En otros casos las actividades realizadas son de carácter crítico, como se 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 ocacionar 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 conputacionales, 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 diseadores 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 computacion *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 asi la nosiva 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

Model Checking

2.1. El proceso de Model Checking

- BDD ... SAT

2.2. NuSMV

- NuSMV como funciona, capacidades, ventajas de haberlo elegido.

Capítulo 3

Tolerancia a fallas

- Leer algun paper a cerca de eso y rellenar aca.

Capítulo 4

Conocimientos previos

- Con falluto queremos poder escribir un modelo que en nuestra cabeza normalmente esta abstraído a sistemas de transiciones etiquetadas y estructuras de Kripke. Queremos además hacerlo de manera declarativa en cuanto a la inserción de fallas, sin tener que implementar la funcionalidad de las mismas.

4.1. LTS

4.2. Kripke Structures

Capítulo 5

El lenguaje de Falluto2.0

5.1. Lenguaje general

- Como logra 'describir claramente la red de automatas' de manera intuitiva.

5.2. Inserción de fallas

Capítulo 6

Compilación de Falluto2.0

6.1. transiciones normales y transiciones de falla

6.2. fairness

6.3. propiedades comunes

Capítulo 7

Casos de estudio

7.1. Commit atómico

7.2. Ejércitos bizantinos ?

7.3. Filósofos comensales ?

7.4. Falla bizantina ?

7.5. Elección de lider ?

7.6. Protocolos lamport

7.7. Feedback de contacto de Naza

7.8. Red satelital ultra secreta?

Capítulo 8

Conclusión

Capítulo 9

Apéndice A - Manual de Falluto2.0

Capítulo 10

Apéndice B - Sintáxis formal de Falluto2.0

Capítulo 11

Apéndice C - Ejemplo paradigmático de compilación.

Capítulo 12

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 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 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; Nicols 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