Prototipado en FPGAs para inyección de fallas. Aplicación a sistemas distribuidos sobre bus CAN

Tesis de maestría en Ingeniería Eléctrica

Julio PÉREZ ACLE

Director de Tesis:

Matteo SONZA REORDA, Politecnico di Torino, Italia

Director Académico:

Rafael CANETTI, IIE, Facultad de Ingeniería, Universidad de la República, Uruguay

Tribunal:

Rafael CANETTI, IIE, Facultad de Ingeniería, Univ. de la República, Uruguay

Marcelo LUBASZEWSKI, Univ. Federal do Rio Grande do Sul, Brasil

Gregory RANDALL IIE, Facultad de Ingeniería, Univ. de la República, Uruguay

Raúl VELAZCO, INPG, Laboratoire TIMA, Grenoble, Francia

Instituto de Ingeniería Eléctrica

Facultad de Ingeniería

Universidad de la República

Montevideo, Uruguay

8 de setiembre de 2005

ISSN : 1510-7264 - Reporte Técnico Nº

Contenido

Introducción

Existen cada día más sistemas críticos en los que la indisponibilidad del servicio u otro malfuncionamiento provocado por fallas puede tener consecuencias muy graves, tanto desde el punto de vista económico como en eventuales daños a las personas o al medio ambiente.

En el diseño y la operación de sistemas de este tipo deben tomarse precauciones para minimizar la cantidad de salidas de servicio y malfuncionamientos. Esto puede hacerse por la vía de minimizar la cantidad de fallas, por la vía de hacer que el sistema sea capaz de seguir brindando el servicio en presencia de fallas, o por la vía de minimizar las consecuencias de un servicio deteriorado.

Tiene interés entonces conocer en primera instancia a qué fallas estará expuesto un sistema dado y con qué frecuencia se producirán, y en una segunda instancia evaluar cuál será el comportamiento del sistema frente a esas fallas. El objetivo de un análisis de este tipo varía a lo largo del ciclo de vida del sistema: en la fase de desarrollo será útil para depurar y mejorar los mecanismos de tolerancia a fallas. En la fase de producción servirá para verificar si se han alcanzado o no los niveles requeridos de confiabilidad y justificar así la confianza que pueda depositarse en el servicio brindado por el sistema. Para realizar esta evaluación existen métodos analíticos y métodos experimentales.

Las técnicas de inyección de fallas son métodos experimentales que consisten en analizar el comportamiento de un modelo o un prototipo del sistema frente a fallas provocadas artificialmente. En particular en sistemas que involucren circuitos electrónicos digitales, que son los que nos ocuparán en este trabajo, la inyección de fallas consistirá en modificar por algún mecanismo el valor de señales de entrada o internas del circuito. Las principales ventajas de esta técnica son la repetibilidad de los experimentos y el permitir obtener una cantidad estadísticamente suficiente de experimentos en un tiempo varios órdenes de magnitud menor al necesario si nos limitáramos a observar las fallas producidas naturalmente en el sistema en operación.

La inyección de fallas es una herramienta útil para depurar los mecanismos de tolerancia a fallas durante la fase de desarrollo, para evaluar qué fracción de las fallas a las que se somete el sistema provocarán un malfuncionamiento del mismo y para evaluar la gravedad de los efectos provocados por esos malfuncionamientos. Para estimar otras métricas relacionadas a la confiabilidad hace falta además obtener información sobre la distribución estadística de las fallas.

Por otra parte, la industria automotriz es un área en la que hay una demanda creciente de requerimientos de confiabilidad sobre sistemas electrónicos. Cada vez más funciones críticas para la seguridad (control de frenado, suspensión, volante) están pasando a ser realizadas por unidades electrónicas de control (ECUs) interconectadas a través de una red de datos con una topología de bus. Uno de los estándares más utilizados para esa red de datos a bordo es CAN (Controller Area Network) introducido por Bosch y posteriormente normalizado como el estándar ISO 11898.

El objetivo del presente trabajo fue alcanzar el estado del arte de las técnicas de inyección de fallas y principalmente explorar la utilización de técnicas de prototipado en FPGAs para acelerar campañas de inyección de fallas en sistemas electrónicos digitales.

El área de aplicación elegida fue la de sistemas distribuidos formados por las unidades electrónicas de control de un automóvil interconectadas por un bus CAN. Los controladores del protocolo CAN y el propio bus fueron prototipados en un circuito sintetizado dentro de un único chip.

Se plantearon objetivos en cierto modo contrapuestos: por un lado se procuró analizar el efecto de fallas en el interior del controlador de acceso al bus CAN, para lo que hace falta modelar en detalle el circuito del controlador. Por otro lado se trazó como objetivo analizar el efecto de las fallas a un nivel lo más alto posible, idealmente al nivel del comportamiento del vehículo. De este modo es posible incluir en el análisis el efecto de los mecanismos de tolerancia a fallas intrínsecos del sistema, como puede ser la inercia mecánica en el caso de un vehículo, así como de los mecanismos de tolerancia a fallas introducidos ex-profeso en el diseño de la aplicación y el protocolo de comunicaciones. Este doble objetivo obligó a describir diferentes partes del

sistema con diferentes niveles de abstracción para poder llegar a un modelo computacionalmente accesible.

Resultados

Como resultado de este trabajo se obtuvieron herramientas para simplificar la realización de campañas de inyección de fallas sobre modelos de transferencia de registros de sistemas distribuidos comunicados por un bus CAN. Se desarrollaron dos conjuntos de herramientas que comparten la misma descripción en lenguaje VHDL (Very high speed integrated circuit Hardware Description Language) del bus y de los controladores de acceso al mismo. Por un lado se desarrollaron herramientas para inyección en un simulador VHDL estándar ejecutando en un computador (se utilizó Modelsim), en las que el comportamiento de cada unidad de control se describe como módulos VHDL simulables. Por otro lado, herramientas para inyección de fallas sobre un prototipo hardware en que los controladores de acceso al bus y el propio bus fueron sintetizados en una placa FPGA insertada en un slot PCI de un computador personal, mientras que el comportamiento de las unidades de control es emulado por software.

Un subproducto importante de los trabajos realizados para la presente tesis fue la obtención de la descripción en VHDL sintetizable de un controlador de acceso a bus CAN. Este desarrollo fue necesario porque se pretendió analizar el efecto de fallas sobre los registros internos de los controladores de acceso al bus CAN, para lo que fue necesario tener acceso al código fuente del mismo a efectos de poder introducir las modificaciones requeridas para la inyección de fallas.

Se realizaron varias campañas de inyección de fallas donde se ensayaron las herramientas desarrolladas obteniéndose una muy buena performance en cantidad de fallas inyectadas por unidad de tiempo. Utilizando las herramientas desarrolladas junto con un modelo del comportamiento de un vehículo, desarrollado previamente por otro grupo de investigación del Politecnico di Torino, fue posible analizar los efectos de cada falla a lo largo del sistema y en consecuencia identificar las fallas críticas desde el punto de vista de la seguridad del vehículo.

Los resultados numéricos de los experimentos de inyección de fallas presentados en este trabajo están fuertemente influidos por lo bien o mal que se comporta frente a

fallas el controlador CAN utilizado. Dicho controlador no fue diseñado con el objetivo de hacerlo tolerante a fallas y no hubo un interés especial por hacer una evaluación cuantitativa de sus atributos en cuanto a confiabilidad.

Entonces, los resultados más fuertes del presente trabajo no son los resultados numéricos, que no son generalizables, sino que son por un lado los resultados cualitativos en cuanto a los efectos que puede tener una falla en el controlador CAN sobre el comportamiento de un vehículo, y por otro lado la metodología de modelado en diferentes niveles y las herramientas desarrolladas tendientes a integrar esos diferentes niveles en un mismo experimento.

Cronología

Las tareas para el desarrollo descrito líneas arriba fueron realizadas principalmente durante dos pasantías realizadas a principios de 2003 y a principios de 2004 en el Dipartimento di Automatica e Informatica del Politecnico di Torino. En ambas pasantías trabajé en el seno del grupo dirigido por el tutor de esta tesis Matteo SONZA, en estrecha colaboración con Massimo VIOLANTE del mismo grupo.

Resultados parciales de estos trabajos fueron publicados previamente en memorias de conferencias [18] [20] [21] [22] [19].

Organización del documento

El trabajo está organizado en tres partes principales y varios anexos.

En la Parte I se presenta la terminología y un resumen de los principales conceptos de *dependabilidad* [Capítulo 1] y los antecedentes en técnicas de inyección de fallas [Capítulo 2].

En la Parte II se hace una breve presentación del área de aplicación elegida, más precisamente del protocolo CAN [Capítulo 3] y de las redes utilizadas en la industria automotriz [Capítulo 4].

En la Parte III se describen las herramientas desarrolladas y los experimentos realizados. En [Capítulo 5] se presenta el controlador de protocolo CAN desarrollado, en [Capítulo 6] el ambiente para permitir la inyección de fallas y en [Capítulo 7] las

campañas experimentales realizadas. Algunas conclusiones y líneas de trabajo futuro se resumen en [Capítulo 8].

PARTE I

DEPENDABILIDAD Y TÉCNICAS DE INYECCIÓN DE FALLAS

En esta parte se introducen algunos conceptos básicos para las técnicas de inyección de fallas.

En el Capítulo 1 se presentan términos y definiciones relativos a la capacidad de los sistemas de proveer un servicio en el cual se pueda confiar en forma justificada, y a los mecanismos con que cuentan los diseñadores y usuarios de esos sistemas para fundamentar esa confianza. Esa propiedad de un sistema se denomina en inglés *dependability*. En el Anexo 1 se presenta un resumen de la terminología utilizada en este trabajo y su correspondencia con los términos utilizados en otros idiomas.

La disciplina de inyección de fallas es introducida en el Capítulo 2. Se presentan las definiciones básicas, las diferentes finalidades perseguidas al realizar experimentos de inyección de fallas y los principales antecedentes en el área. Se pone el énfasis en la inyección de fallas en circuitos electrónicos digitales, y en particular en las técnicas que se utilizaron en el presente trabajo: inyección de fallas en simulaciones de circuitos descritos a nivel de transferencia de registros e inyección de fallas en prototipos de los circuitos sintetizados dentro de un FPGA.

Capítulo 1

Dependabilidad. Un resumen de los principales conceptos

Existen una serie de términos para referirse a conceptos relacionados con la confiabilidad de los sistemas que, por tener significados similares entre sí en lenguaje natural, dieron durante mucho tiempo lugar a confusión y superposiciones. Durante los años 80 hubo varios trabajos que fueron formando una estructura de conceptos y terminología consistentes. Esta estructura fue sintetizada en un trabajo de Laprie [30] y varias actualizaciones posteriores [10] [24] [53][5].

La *dependabilidad* de un sistema se define como su capacidad de proveer un servicio en el cual se puede confiar en forma justificada. Es un concepto cualitativo que integra diferentes aspectos o atributos cuantificables (confiabilidad, disponibilidad, seguridad, etc.). El desempeño de los sistemas en estos aspectos no es el ideal debido a la existencia de fallas, errores y malfuncionamientos (denominados amenazas a la *dependabilidad* en la terminología de Laprie). Para mejorar la *dependabilidad* se procura en forma combinada por un lado evitar, tolerar y eliminar las fallas y por otro lado estimar la cantidad y posibles consecuencias de las mismas. A estas técnicas se les llama en conjunto los medios para alcanzar la *dependabilidad*.

Las técnicas de inyección de fallas consisten en analizar el comportamiento real o simulado de un sistema en presencia de fallas provocadas artificialmente. Estas técnicas se utilizan con diferentes finalidades: en la etapa de desarrollo para depurar, corregir o verificar los elementos y mecanismos que pretenden hacerlo tolerante a fallas; más adelante para caracterizar y cuantificar los malfuncionamientos provocados por las fallas que escapan a estos mecanismos, contribuyendo así a justificar la confianza depositada en el servicio.

Se presenta en lo que resta de este Capítulo 1 un resumen de la terminología presentada por Laprie, poniendo énfasis en los aspectos más relevantes para las técnicas de inyección de fallas desarrolladas en el resto del trabajo. Dado que en su mayor parte se trata de definiciones se ha preferido hacer una traducción casi textual

de las partes de textos de Laprie [24] [53] que más interesan para nuestro trabajo. En el Anexo 1 se da un glosario de términos y su correspondencia con los términos en inglés utilizados en los trabajos originales.

1.1. Definiciones

Una exposición sistemática de los conceptos de *dependabilidad* consiste de tres partes: las **amenazas** a, los **atributos** de, y los **medios** por los cuales la *dependabilidad* es obtenida.

Los sistemas de cómputo se caracterizan por cuatro propiedades fundamentales: funcionalidad, performance, costo, y *dependabilidad*. La *dependabilidad* es la capacidad de proveer un servicio en el cual se puede confiar en forma justificada. El **servicio** que provee un sistema es su comportamiento tal como es percibido por sus usuarios; un **usuario** es otro sistema (físico, humano) que interactúa con el primero en la **interfaz del servicio**. La **función** de un sistema es aquello para lo cual fue creado, y está descrita en la especificación funcional del mismo.

1.2. Las amenazas: Fallas (faults), Errores (errors), y Malfuncionamientos (failures)

Se entrega un servicio correcto cuando el servicio implementa la función del sistema.

Un **malfuncionamiento** del sistema es un evento que ocurre cuando el servicio provisto se desvía del servicio correcto. Un sistema puede malfuncionar ya sea porque no cumple con la especificación o porque la especificación no describe adecuadamente la función. Un malfuncionamiento es una transición desde servicio correcto a servicio incorrecto, i.e., a no implementar la función del sistema. Una transición desde servicio incorrecto a servicio correcto es una **restauración de servicio**. El intervalo de tiempo durante el cual se entrega servicio incorrecto es una **salida de servicio**.

Un **error** es un estado del sistema que puede provocar un subsiguiente malfuncionamiento: cuando un error llega a la interfaz del servicio y altera el servicio suministrado sucede un malfuncionamiento. Una **falla** es la causa hipotética o adjudicada de un error. Una falla está **activa** cuando produce un error, de lo contrario está **dormida**.

Un sistema no siempre malfunciona de la misma manera. Las maneras en que un sistema puede malfuncionar son sus **modos de malfuncionamiento.** Los modos de malfuncionamiento caracterizan al servicio incorrecto desde tres puntos de vista: a) el dominio del malfuncionamiento (de valor, de temporización), b) la percepción del malfuncionamiento por los diferentes usuarios del sistema (consistentes o no), y c) las consecuencias del malfuncionamiento en el ambiente.

Un sistema consiste de un conjunto de componentes que interactúan, por lo tanto el estado del sistema es el conjunto de los estados de sus componentes. Inicialmente una falla provoca un error en el estado de uno o varios componentes, pero no se produce un malfuncionamiento del sistema hasta tanto el error no llega a la interfaz del servicio brindado por el sistema. Una forma conveniente de clasificar los errores es describirlos en términos de los malfuncionamientos que provocan, usando la terminología introducida más arriba: errores de valor vs. errores de temporización; errores consistentes o inconsistentes ('Bizantinos') cuando la salida va a más de un componente; errores de diferente severidad: menores, ordinarios, catastróficos. Un error es **detectado** si su presencia en el sistema es indicada por un mensaje o señal de error originada en el interior del sistema. Errores que están presentes pero no han sido detectados son **errores latentes**.

Las fallas y sus orígenes son muy variados. Posibles criterios de clasificación de las mismas son:

- La fase de creación o ocurrencia de la falla (fallas de desarrollo, fallas operativas)
- Las fronteras del sistema (fallas internas o externas)
- Fallas de hardware o de software
- Causa fenomenológica (Fallas naturales o humanas)
- Intencionalidad (Fallas deliberadas o no deliberadas)
- Persistencia (Fallas permanentes o transitorias)

Se puede argüir que introducir las causas fenomenológicas como criterio de clasificación puede conducir recursivamente a preguntas como "¿y por qué los programadores cometen errores?", "¿por qué fallan los circuitos integrados?". El concepto falla sirve para detener la recursión. Por eso la definición dada: causa *hipotética o adjudicada* de un error. Esta causa adjudicada o hipotética puede variar dependiendo del punto de vista elegido: mecanismos de tolerancia a fallas, ingeniero de mantenimiento, centro de reparaciones, desarrollador, físico en semiconductores, etc..

A la relación entre fallas, errores, y malfuncionamientos descrita líneas arriba se le llama **patología de las fallas,** y se resume en la Fig. 2, que muestra la cadena fundamental de propagación de las amenazas a la *dependabilidad*. Los arcos en esa cadena expresan una relación de causalidad entre fallas, errores, y malfuncionamientos.

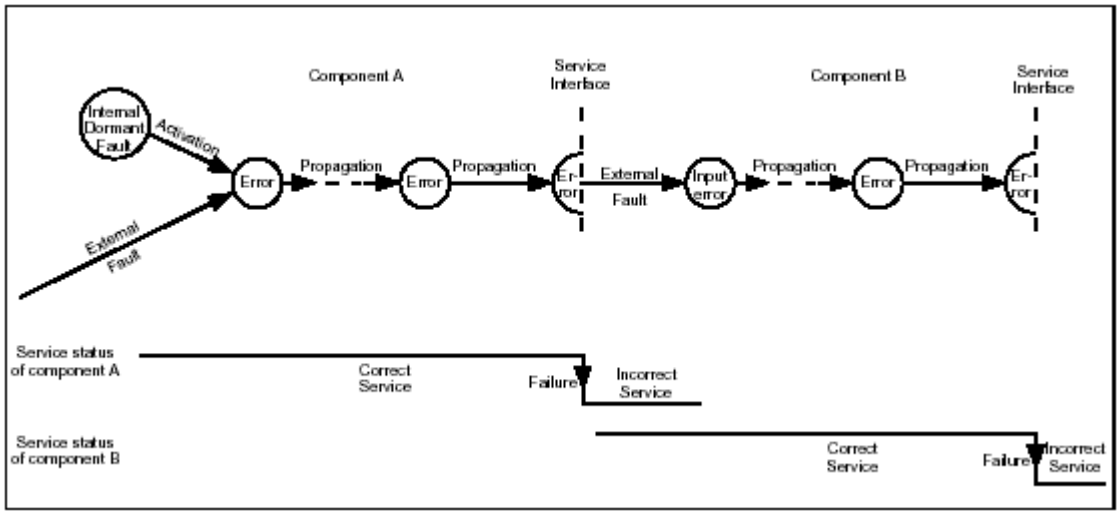

Fig. 1 Propagación de errores. (tomada de [53])

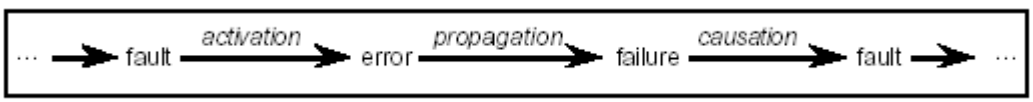

Fig. 2 Cadena fundamental de las amenazas a la dependabilidad (tomada de [53])

1.3. Los Atributos de la dependabilidad

La Dependabilidad es un concepto integrador que abarca los siguientes atributos: **disponibilidad**: estar listo para proveer servicio correcto; **confiabilidad**: continuidad

de servicio correcto; **seguridad-safety**: ausencia de consecuencias catastróficas sobre los usuarios y el ambiente; **confidencialidad**: ausencia de difusión no autorizada de información; **integridad**: ausencia de alteraciones no adecuadas del estado del sistema; **mantenibilidad**: aptitud para ser sometido a reparaciones y modificaciones.

Cada uno de los atributos listados arriba puede ser requerido con más o menos énfasis dependiendo de la aplicación: siempre se requiere disponibilidad, si bien en grados diversos, mientras que confiabilidad, seguridad-safety, y confidencialidad pueden o no ser requeridos. El grado en el cual un sistema posee los atributos de *dependabilidad* debería ser interpretado en un sentido relativo, probabilístico y no en un sentido absoluto, determinístico: debido a la inevitable presencia u ocurrencia de fallas, los sistemas no son nunca totalmente disponibles, confiables o seguros.

Las variaciones en el énfasis puesto en los diferentes atributos de la *dependabilidad* afectan directamente cual es el adecuado balance de técnicas (**prevención**, **tolerancia**, **remoción** y **predicción de fallas**) a ser empleadas para hacer que el sistema resultante sea *dependable*.

Este problema es aún mas grave ya que algunos de los atributos coliden (e.g. disponibilidad y seguridad-safety, disponibilidad y seguridad-security), haciendo necesarios compromisos de diseño.

1.4. Los Medios para alcanzar la *dependabilidad*

El desarrollo de un sistema de cómputo *dependable* requiere la utilización combinada de un conjunto de cuatro técnicas: **prevención de fallas**: cómo evitar la ocurrencia o introducción de fallas; **tolerancia a fallas:** cómo proveer servicio correcto en presencia de fallas; **remoción de fallas:** cómo reducir el número o la severidad de las fallas; y **predicción de fallas:** cómo estimar el número actual, la incidencia futura y las posibles consecuencias de las fallas.

Prevención de fallas

La prevención de fallas se obtiene empleando técnicas de buen diseño y de control de calidad. Estas son técnicas generales de ingeniería y exceden ampliamente la disciplina de la *dependabilidad*, por lo que no nos extenderemos sobre este tema.

Tolerancia a fallas

La tolerancia a fallas intenta mantener la entrega de un servicio correcto en presencia de fallas activas. En general se implementa por medio de la detección de errores y la posterior recuperación del sistema.

La **detección de errores** origina un mensaje o señal de error en el interior del sistema. Un error presente que no ha sido detectado es un error latente. Existen dos clases de técnicas para la detección de errores: (a) detección de errores concurrente, que tiene lugar durante la entrega del servicio; y (b) detección de errores "preemptiva", que tiene lugar con la entrega del servicio suspendida; chequeando el sistema en busca de errores latentes y fallas dormidas.

La **recuperación del sistema** lo transforma desde un estado que contiene uno o más errores y (posiblemente) fallas a un estado sin fallas y errores detectados que puedan volverse a activar.

La recuperación consiste en el manejo de errores y de fallas.

El **manejo de errores** elimina errores del sistema. Puede tomar dos formas: (a) **rollback**, en que la transformación del estado del sistema consiste en llevar al sistema a un estado previamente almacenado, anterior a la detección del error; este estado almacenado es un **checkpoint**, (b) **rollforward**, donde el estado sin errores detectados es un estado nuevo.

El **manejo de fallas** evita que las fallas localizadas sean activadas nuevamente. El manejo de fallas involucra cuatro pasos: a) **diagnóstico**, que identifica y registra la o las causas de errores en términos de ubicación y tipo. b) **aislación** de la falla, que excluye física o lógicamente a los componentes en falla de participar en el suministro del servicio, i.e., se hace que la falla se convierta en dormida. c) **reconfiguración** del sistema, que o bien introduce componentes de reserva o reasigna las tareas sobre los componentes que no han fallado. d) **reinicialización** del sistema, que chequea, actualiza y registra la nueva configuración. Usualmente a continuación de este manejo de fallas se realiza un mantenimiento correctivo que elimina la falla aislada. El factor que distingue la tolerancia a fallas del mantenimiento es que el mantenimiento requiere la participación de un agente externo.

El disponer de redundancia suficiente permite recuperar al sistema sin una detección de errores explícita. A esta forma de recuperación se le llama **enmascaramiento de fallas**.

Remoción o eliminación de fallas

La remoción de fallas se realiza tanto durante la fase de desarrollo como durante la vida operativa de un sistema.

Durante la fase de desarrollo de un sistema, la remoción de fallas consta de tres pasos: verificación, diagnóstico y corrección. **Verificación** es el proceso de controlar si el sistema cumple las llamadas condiciones de verificación. Si no las cumple entonces corresponde proceder con los otros dos pasos: **diagnosticar** las fallas que impidieron cumplir las condiciones y luego realizar las **correcciones** necesarias.

Las técnicas de verificación pueden clasificarse dependiendo de si involucran o no ejercitar el funcionamiento del sistema. A la verificación de un sistema sin una ejecución del mismo se le llama **verificación estática**. Verificar un sistema a través de una ejecución del mismo es una **verificación dinámica**. La ejecución puede ser simbólica o puede ser una ejecución real con entradas reales aplicadas al sistema en lo que se llama **test de verificación** (o **test** a secas) del sistema. Un aspecto importante es la verificación de los mecanismos de tolerancia a fallas, ya sea a) verificación estática formal, y b) tests en los cuales los patrones de test deben contener fallas o errores, a esto último se le denomina **inyección de fallas**.

La remoción de fallas durante la vida operativa del sistema es el **mantenimiento**, ya sea preventivo o correctivo.

Predicción de fallas

La predicción de fallas es llevada a cabo realizando una evaluación del comportamiento del sistema con respecto a la ocurrencia o activación de fallas. Esta evaluación puede realizarse de dos formas: (1) **evaluación cualitativa u ordinal**, que procura identificar, clasificar y ordenar los modos de malfuncionamiento; (2) **evaluación cuantitativa o probabilística**, que procura evaluar en términos probabilísticos el grado de satisfacción de determinados atributos de la

dependabilidad; estos atributos pueden verse entonces como medidas de la *dependabilidad*.

La alternancia entre servicio correcto y servicio incorrecto se cuantifica para definir confiabilidad, disponibilidad y mantenibilidad como medidas de la *dependabilidad*: (1) **confiabilidad**: una medida de la provisión continuada de servicio correcto, o en forma equivalente, el tiempo hasta la falla; (2) **disponibilidad**: una medida de la provisión de servicio correcto con respecto a la alternancia "servicio correcto-servicio incorrecto"; (3) **mantenibilidad**: medida del tiempo hasta la restauración del servicio desde la última falla, o en forma equivalente, provisión continuada de servicio incorrecto. (4) **seguridad-safety:** es una extensión de confiabilidad: Si se agrupan bajo un estado "seguro" los estados de servicio correcto y los estados de servicio incorrecto debidos a fallas no catastróficas, entonces puede definirse seguridad-safety como el tiempo hasta malfuncionamientos catastróficas.

Las dos principales maneras de atacar la evaluación probabilística de las medidas de *dependabilidad* son el **modelado** y el **test de evaluación**. Cuando se evalúan sistemas tolerantes a fallas, la cobertura provista por los mecanismos de manejo de fallas y errores tienen una importancia fundamental. La evaluación de esta cobertura también puede realizarse por modelado o por testing, usualmente llamado en este caso **inyección de fallas**.

Finaliza aquí el resumen de la terminología de *dependabilidad* introducida por Laprie. En el próximo apartado se presentan los antecedentes y las técnicas más usuales de inyección de fallas.

Capítulo 2

Técnicas de inyección de fallas

La inyección de fallas consiste en analizar el comportamiento de un modelo del sistema en presencia de fallas provocadas artificialmente. Usualmente los sistemas están modelados con diferentes niveles de abstracción [9][17] en cada una de las etapas del proceso de desarrollo. Así, en las etapas iniciales el sistema puede estar modelado con cadenas de Markov, redes de Petri o algún otro modelo analítico; más adelante se puede utilizar un modelo en nivel de transferencia de registros (RTL) para simular el sistema o, si el modelo es sintetizable, cargarlo en un FPGA y emular su funcionamiento; finalmente pueden inyectarse las fallas físicamente sobre un prototipo o un ejemplar del producto final. En este trabajo se ha incursionado en la inyección de fallas sobre modelos RTL del sistema, tanto sobre una simulación como sobre el circuito sintetizado en un FPGA.

A continuación se presentan los diferentes objetivos perseguidos al realizar experimentos de inyección de fallas. Luego se introduce el modelo FARM, una abstracción comúnmente utilizada para especificar y describir experimentos de inyección de fallas, y finalmente se hace un rápido repaso de ventajas, desventajas y principales antecedentes para cada uno de los mecanismos de inyección utilizados comúnmente

2.1. Finalidades de la inyección de fallas

La inyección de fallas se utiliza con finalidades de ayuda al diseño y de verificación [10] [9] [17].

En etapas tempranas del desarrollo de un sistema es una herramienta útil para ejercitar los mecanismos de tolerancia a fallas en presencia de las fallas que supuestamente deben soportar. Esto permite detectar tempranamente imperfecciones de los mecanismos de tolerancia a fallas y de los procedimientos de test, e iniciar en consecuencia un ciclo de corrección y rediseño. En este contexto la inyección de fallas es una herramienta de **ayuda al diseño**, facilitando la **detección y remoción de**

fallas de diseño en los mecanismos de tolerancia a fallas y en los procedimientos de test.

La finalidad en este caso es identificar las fallas que escapan a los mecanismos de detección, caracterizar la gravedad de las consecuencias acarreadas por ellas y recoger información adicional que pueda ser útil para el rediseño. Es fundamental la repetibilidad de los experimentos, para verificar luego de introducida la corrección si esta fue efectiva.

La inyección de fallas es también una herramienta importante para pronosticar la frecuencia y gravedad de los malfuncionamientos que tendrá el sistema durante su operación (**predicción de fallas**). A esto contribuye ya que permite medir la **cobertura** de los procedimientos de verificación y los mecanismos de tolerancia a fallas con respecto al tipo de fallas inyectadas. La cobertura de un test es la fracción de las fallas que son puestas en evidencia por el mismo. El concepto es análogo para el caso de los mecanismos de tolerancia a fallas. Otro resultado de la inyección de fallas es la obtención de medidas de parámetros de funcionamiento del sistema, como los tiempos de latencia de errores y tiempos de activación de fallas.

La evaluación de la cobertura de los procedimientos de test y de los mecanismos de tolerancia de fallas por sí sola no es suficiente para evaluar otros atributos de la *dependabilidad*. Para estimar parámetros cuantitativos como tiempo medio entre fallas, disponibilidad, confiabilidad, hace falta además tener conocimiento de la distribución estadística de las fallas que se producen.

2.2. Espacio de fallas (Fault Space)

Un enfoque interesante es ver a cada una de las fallas que pueden afectar a un sistema como puntos en un espacio multidimensional al que se le llama **espacio de fallas** (**FS**). Las dimensiones del espacio de fallas incluyen características temporales como el instante de ocurrencia y la duración de la falla (cuándo), indicación de cuál es el tipo de falla producido y eventualmente el valor asociado a la misma (cómo) y la ubicación dentro del sistema del elemento en el cual se produce la falla (dónde).

El espacio de fallas definido así busca modelar todas las fallas que pueden ocurrir en el sistema. Sin embargo es muy difícil obtener un modelo de fallas que sea completo

en el sentido de cubrir todas las fallas posibles, y es también muy difícil demostrar que un modelo dado es completo. Por otra parte la dificultad y duración de un experimento de inyección de fallas evidentemente crece al ampliar el espacio de fallas.

Otro factor a tener en cuenta es la distribución estadística de la ocurrencia de cada tipo de falla. Si bien esta distribución es a menudo difícil de estimar, en muchos casos puede afectar fuertemente los resultados obtenidos.

Debido a la dificultad en estimar todos estos factores, a menudo se toman soluciones de compromiso con respecto al espacio de fallas a considerar. Sin embargo esto no debe olvidarse a la hora de valorar la aplicabilidad de los resultados a determinado sistema.

2.3. Modelo FARM

El modelo FARM [9][17] para caracterizar a una serie de experimentos de inyección de fallas fue desarrollado en LAAS-CNRS a comienzos de los años 90. En este modelo los experimentos se describen a partir de cuatro conjuntos de entradas o salidas del sistema: el conjunto F (*faults*) lista las fallas que se van a inyectar al sistema; el conjunto A (*activation*) consiste en el conjunto de patrones de entrada suministrados al sistema bajo prueba; en cada experimento se inyecta una falla y se registra un conjunto R (*readouts*) de salidas; y finalmente a partir del análisis de los elementos de los tres conjuntos anteriores se deduce un conjunto M (*measures*) de medidas.

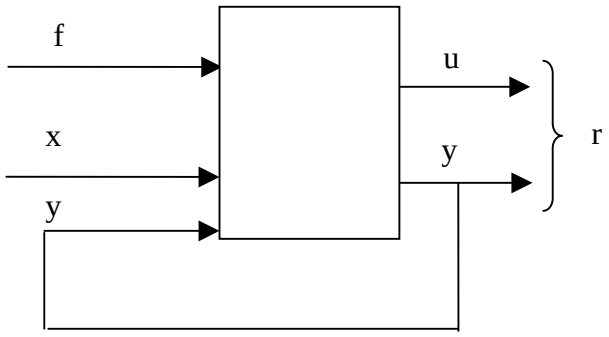

Fig. 3 Diagrama de bloques mostrando explícitamente a las fallas como una entrada.

En el diagrama de bloques de la Fig. 3 se ha extendido el diagrama tradicional de una máquina de estados finita para indicar explícitamente a la falla a inyectar como una entrada.

El conjunto F es la lista de fallas a inyectar durante la serie de experimentos. Es sólo un subconjunto de las fallas posibles, seleccionado dentro del espacio de fallas FS que se haya modelado para el sistema.

La activación del sistema queda determinado por la secuencia de entradas externas *x(t)* aplicadas y el valor inicial *yo* del estado *y* del sistema en *t = 0*, al inicio del experimento. Junto con las características del sistema bajo ensayo determina si una falla dada se activa produciendo un error o no, de ahí su denominación. El conjunto A está formado por la lista de las diferentes duplas *(x(t), yo)* que se utilicen en los experimentos.

En el conjunto R se registran tanto salidas externas ***u*** como estados ***y***.

En el sistema ampliado para considerar explícitamente las fallas, una vez elegida la terna *(x(t), yo, f)* queda determinada la evolución de las salidas *u(t)* y de los estados *y(t)*. El comportamiento del sistema puede verse como una transformación *T* entre un dominio de entrada formado por las posibles ternas *(x(t), yo, f)* y un dominio de salida formado por las correspondientes trayectorias de salidas y estados *(u(t), y(t))*. Cada terna determina un posible experimento de inyección de fallas.

$T\{ (x(t), yo, f) \} = (u(t), y(t))$

A menudo para un conjunto de activación dado $(x_a(t), yo_a)$, se toma como referencia el experimento en el cual no se inyecta falla alguna. A este experimento se le llama "*golden run*". Una falla inyectada es activada durante un experimento si la trayectoria en el dominio de salida difiere de la registrada durante el experimento de referencia:

$T(x_a(t), yo_a, f) != T(x_a(t), yo_a, fg)$; donde fg denota "ausencia de falla".

Una diferencia con la *golden run* indica que se produce un error, pero no necesariamente un malfuncionamiento del sistema. Dicho de otra forma, diferentes trayectorias en el dominio de salida pueden ser igualmente aceptables y mantener el servicio brindado por el sistema dentro de lo especificado. Cuando en una serie de

experimentos de inyección de fallas se clasifican las fallas de acuerdo al modo de malfuncionamiento que provocan, el conjunto M incluye predicados o aseveraciones cuyo resultado lógico indica si durante el experimento se produjo o no determinado malfuncionamiento [11] [41]. Este conjunto de aseveraciones en definitiva describe los *modos de malfuncionamiento* del sistema. Su elaboración es difícil de generalizar y exige conocimiento y análisis específico del sistema bajo prueba. Por ese motivo es una parte importante del esfuerzo de preparación de una serie de experimentos de inyección de fallas y es una de las principales dificultades para automatizar la generación de experimentos de inyección de fallas.

2.4. Estimación estadística de cobertura de fallas

En este apartado se incluye una muy breve presentación de la estimación estadística de la cobertura de fallas a partir de experimentos de inyección, y en particular de la influencia de la distribución de probabilidad con que se seleccionan las fallas dentro del espacio de fallas sobre la estimación de la cobertura. Un análisis más completo puede encontrarse en [17].

Cuando se utiliza inyección de fallas para realizar un análisis cuantitativo, el parámetro básico a calcular es la cobertura de los mecanismos de tolerancia a fallas. Esta se define como la siguiente probabilidad condicional:

$$C = P(\textit{Manejo adecuado de la falla} \mid \textit{Ocurrió una falla del espacio FS})$$

Representemos la ocurrencia de una falla con una variable aleatoria F con recorrido en el espacio de fallas FS, excluido el caso “ausencia de falla”. Entonces la sumatoria de $P(F = f)$ en todas las fallas f del espacio de fallas FS vale 1. Representemos también al evento “manejo adecuado de una falla” con otra variable aleatoria binaria Y ($Y = 1$ si la falla se manejó de manera adecuada y $Y = 0$ en caso contrario). Entonces la cobertura puede expresarse como el valor esperado de la variable Y:

$$C = \sum_{f \in FS} P(Y = 1 \mid F = f).P(F = f)$$

$$C = \sum_{f \in FS} y(f).P(F = f)$$

El primer factor valdrá 0 o 1 dependiendo de si la falla *f* está cubierta, de manera que para un sistema dado este factor puede representarse por una función fija *y(f)*.

El segundo término representa la distribución de probabilidad de ocurrencia de las fallas dentro del espacio de fallas.

Si al realizar un experimento de inyección de fallas seleccionamos las mismas con una distribución de probabilidades diferente, entonces tendremos una nueva variable aleatoria *F'* con su correspondiente distribución *P(F'= f)*. Se debe introducir una variable aleatoria binaria *Y'* diferente de *Y* para describir el evento "manejo adecuado de la falla durante el experimento de inyección de fallas". Al valor esperado *C'* de esta nueva variable aleatoria se le llama **cobertura forzada**.

$$C' = E[Y']$$

$$C' = \sum_{f \in FS} y(f).P(F' = f)$$

La distribución de *F'* es elegida al diseñar el experimento de inyección de fallas, mientras que la distribución de *F* es una característica del sistema bajo prueba. Tiene interés entonces estudiar la relación entre *C* y *C'* para ayudar a estimar el valor de *C* a partir de los resultados experimentales. En [17] se demuestra que:

$$C = C' + \rho$$

Dónde ρ es la covarianza entre la variable aleatoria *Y'* y el cociente entre la probabilidad real y la probabilidad forzada *P(F = f) / P(F' = f)*, definido en cada punto del espacio FS. En [17] también se demuestra que el valor esperado de ese cociente es 1.

El valor de ρ puede ser positivo o negativo, y en consecuencia la cobertura forzada puede ser mayor o menor que la cobertura real.

Intuitivamente, la cobertura para la distribución de fallas real será mayor que la cobertura forzada (covarianza ρ positiva) si el mecanismo de tolerancia a fallas cubre mejor las fallas que son más probables en la distribución real que las fallas que son más probables en la distribución forzada. Esto muestra la importancia de modelar en

forma adecuada el espacio de fallas y su distribución de probabilidad ya que de lo contrario el resultado obtenido (la cobertura forzada) puede diferir de la cobertura de fallas reales tanto en más como en menos.

2.5. Inyección de fallas por mecanismos Hardware

En la inyección de fallas por mecanismos hardware se dispone de un prototipo real del sistema en el cual se provocan fallas por medios físicos y se analiza el comportamiento posterior del mismo. En la mayoría de los casos la inyección de fallas por mecanismos hardware involucra la utilización de equipamiento de test externo para permitir la inyección de fallas y el registro y análisis de sus efectos. Por ese motivo este mecanismo suele ser más costoso que otros mecanismos.

Dado que se aplica sobre un prototipo o sobre un ejemplar funcionante del producto final, este mecanismo se utiliza en general en la fase de producción.

Se distinguen dos tipos de mecanismos [2]:

- Mecanismos hardware con contacto. En estos mecanismos el dispositivo inyector entra en contacto físico directo con el circuito bajo prueba. Ejemplos de estos mecanismos son la inyección de fallas a nivel de pines de circuito integrado, puntos de prueba en el circuito impreso o contactos de conector de una plaqueta.

- Mecanismos hardware sin contacto. En este caso el inyector no tiene contacto físico directo con el circuito, sino que a través de alguna fuente externa produce algún fenómeno físico que induce efectos parásitos dentro del circuito que modifican su comportamiento. Algunos fenómenos físicos utilizados son el bombardeo con partículas pesadas y la interferencia electromagnética.

En el caso de inyección con contacto los modelos de fallas más utilizados son las fallas “stuck-at”, puentes entre contactos adyacentes, desconexión, etc.

Se utilizan técnicas de *forzado* o de *inserción*. En las técnicas de forzado el inyector se conecta directamente al terminal del circuito integrado o conector de que se trate sin desconectar nada en el circuito bajo prueba. En este caso la impedancia de salida del inyector debe ser muy baja de manera que el nivel lógico impuesto por el inyector

“domine” al nivel lógico impuesto por el circuito. Esto puede provocar corrientes mayores a las garantizadas en las salidas bajo prueba, con el consiguiente riesgo de dañar el circuito.

En las técnicas de inserción el dispositivo inyector se intercala entre salida y entrada. La complejidad mecánica y la manipulación necesaria para preparar un experimento son mayores, pero a cambio se disminuye fuertemente el riesgo de dañar el circuito.

Entre las ventajas de los mecanismos hardware de inyección de fallas podemos distinguir las siguientes:

- los experimentos se ejecutan a la velocidad real del sistema, por lo general mucho mayor que la velocidad de simulación.
- el sistema bajo test es por lo general el producto final o casi final. Por lo tanto se puede detectar y corregir las fallas de diseño introducidas en las fases finales de desarrollo y producción.

La lista de desventajas por su parte incluye:

- requiere equipamiento de test costoso.
- en el caso de inyección con contacto el riesgo de dañar el circuito es importante.
- con el crecimiento sostenido de los niveles de integración, la cantidad de puntos del circuito accesibles ya sea para observar o controlar (inyectar) se hace muy limitada.
- en el caso de inyección por interferencia electromagnética o por bombardeo con partículas pesadas es muy limitado el control sobre el instante y la ubicación exacta de la falla, y por lo tanto la repetibilidad de los resultados de un experimento.

Existen numerosos ejemplos de dispositivos diseñados para inyección de fallas a nivel de pines. Uno de los más conocidos es MESSALINE [9], un sistema para inyección de fallas a nivel de pines por la técnica de forzado desarrollado en LAAS-CNRS,

Francia. Un ejemplo más reciente es AFIT (Advanced Fault Injection Tool) [16] desarrollado por la Universidad Politécnica de Valencia, España.

2.6. Inyección de fallas por mecanismos Software

Si el sistema bajo análisis involucra a un microprocesador entonces en muchos casos es conveniente utilizar el propio procesador del sistema para inyectar fallas o registrar sus efectos. De esta forma se puede realizar inyección de fallas implementada por software, técnica a menudo referida por la sigla en inglés SWIFI (SoftWare Implemented Fault Injection). Se utiliza este mecanismo tanto para inyectar fallas hardware (p. ej. introducir un bit-flip en un registro para emular una falla del tipo Single Event Upset) como fallas software (p. ej. la introducción de bugs por la vía de modificar el código).

La principal ventaja de la inyección de fallas por mecanismos software es el reducido costo, ya que no requiere equipamiento de test especial para su realización. Permite además tener acceso a detalles internos de la aplicación o del sistema operativo totalmente impensables con otros mecanismos de inyección de fallas.

Las desventajas de este mecanismo incluyen:

- la observabilidad y controlabilidad se limita a los elementos accesibles por el programador. La resolución en el eje de tiempos está limitada a los instantes de comienzo de las instrucciones de lenguaje ensamblador del programa bajo prueba. La ubicación de las fallas que es posible inyectar se reduce a los registros internos del procesador y la memoria externa. No es posible por ejemplo inyectar fallas en los Flip-Flops del bloque de control de la CPU. Tampoco es sencillo inyectar fallas en la memoria caché.

- Es muy difícil introducir fallas permanentes.

- En el caso de aplicaciones de tiempo real la inyección por mecanismos software puede ser inaceptablemente intrusiva, ya que la ejecución del software de inyección ocupa tiempo de CPU y puede por tanto impedir el cumplimiento de deadlines.

Existen varios ejemplos de sistemas que utilizan técnicas SWIFI, tanto para sistemas dedicados como para analizar aplicaciones corriendo en sistemas operativos comerciales. BOND [13] es un sistema para inyección de fallas en aplicaciones corriendo sobre sistemas operativos Microsoft desarrollado en el Politecnico di Torino, Italia. XCEPTION [14] es una herramienta SWIFI desarrollada en la Universidad de Coimbra, Portugal. MAFALDA [12] está orientada a aplicaciones de tiempo real con microkernels comerciales, desarrollado en LAAS-CNRS, Francia. FERRARI [8] es una herramienta para emular fallas y errores hardware en computadores Unix, desarrollado en la universidad de Texas, Austin, Estados Unidos.

2.7. Inyección de fallas sobre modelos de descripción hardware

Si se dispone de la descripción del sistema en algún lenguaje de descripción hardware (VHDL, Verilog) entonces puede realizarse la inyección de fallas o bien simulando el circuito o bien cargando en un FPGA un prototipo hardware sintetizado a partir del modelo del sistema. Al segundo mecanismo se le denomina a veces inyección de fallas por emulación (emulated fault injection). Se comentan a continuación las dos alternativas.

Inyección basada en Simulación

Para esta técnica de inyección de fallas es necesario disponer de un modelo del sistema bajo prueba que permita simular el comportamiento del mismo en presencia de fallas. Esta simulación puede realizarse en diferentes niveles de abstracción, por lo que esta técnica puede utilizarse en etapas tempranas del desarrollo como ayuda al diseño de los mecanismos de tolerancia a fallas. Se utilizan modelos en VHDL, Verilog u otros lenguajes que pueden ser desarrollados especialmente para la campaña de inyección de fallas o ser parte de las descripciones utilizadas para la especificación o síntesis del sistema.

Las ventajas principales de la inyección de fallas en simuladores VHDL o Verilog son:

- Bajo costo al no ser necesario hardware de propósito específico.

- Puede utilizarse en etapas tempranas del diseño suministrando información útil para el diseñador.

- Alta observabilidad y controlabilidad, es posible observar o alterar el valor lógico de cualquier elemento interno del circuito en cualquier instante.

Como contrapartida, el tiempo de simulación es muy prolongado para cualquier circuito de tamaño razonable y el desarrollo de los modelos a menudo requiere un esfuerzo considerable. Por otra parte la inyección de fallas se realiza sobre la ejecución simulada de un modelo del sistema y no sobre un ejemplar del sistema final, por lo que no podrá activar fallas que se introducen o manifiestan en las etapas posteriores del desarrollo.

Existen varios caminos para inyectar fallas en la ejecución simulada del sistema bajo prueba. Un primer método consiste en modificar el valor lógico de un elemento interno del circuito mediante comandos del simulador utilizado. Esta técnica tiene la ventaja de ser poco intrusiva al no modificar el código del modelo del sistema. Tiene en cambio la desventaja de atar la implementación al simulador utilizado ya que los comandos de simulador son en general propietarios.

Otra alternativa es modificar el código del modelo para inyectar fallas y observar el comportamiento del mismo. En esta línea una posibilidad es el agregado de componentes a la descripción del circuito para permitir modificar (*saboteurs*) y observar (*probes*) el valor lógico de un elemento interno del circuito. Otra alternativa es introducir *mutaciones* en el código fuente de la descripción hardware del circuito. La variedad de mutaciones que se pueden introducir es amplia: reemplazar un componente en una descripción estructural (p. ej. cambiar un AND por un OR) o modificar estructuras de control en una descripción estructural (p. ej. reemplazar una condición por un valor fijo TRUE (stuck-then) o FALSE (stuck-else), perturbar el valor de una señal, etc.).

Por último, en el caso de modelos VHDL una técnica poco invasiva consiste en inyectar las fallas modificando la definición de los tipos de datos y las funciones de resolución de VHDL. De esta forma pueden definirse asignaciones adicionales a la señal que se desea perturbar que predominen sobre las asignaciones normales del diseño original.

Existen varias herramientas para modificar más o menos automáticamente el código de descripción hardware y realizar campañas de inyección de fallas basadas en simulación. MEFISTO [11] es un conjunto de herramientas desarrollado en colaboración por LAAS-CNRS, Francia (MEFISTO-L) y Chalmers University of Technology (MEFISTO-C) que utilizan casi todas las técnicas enumeradas. VFIT [15] es una herramienta desarrollada en la Universidad de Valencia, España, que utiliza las técnicas de mutaciones de código, saboteurs y comandos de simulador.

Durante el trabajo en esta tesis se realizó inyección de fallas basada en simulación utilizando la técnica de saboteurs sobre una red de nodos conectados por un bus CAN como se presenta en la sección *6.2 Ambiente basado en simulación*. Los resultados fueron presentados en [22].

Inyección basada en Emulación

Si el modelo que se dispone es sintetizable, la inyección de fallas puede acelerarse en órdenes de magnitud por la vía de realizar la prueba sobre un prototipo hardware en lugar de en una simulación. Generalmente esto se hace sintetizando el circuito y cargándolo en un FPGA. Una ventaja adicional es que el sistema puede probarse en condiciones más similares a las de operación normal.

Para inyectar fallas sobre el prototipo hardware es necesario no solamente agregar los mecanismos para observar y perturbar las señales internas sino también para controlar el instante en que se inyecta la falla. Debido a eso las modificaciones que es necesario introducir al circuito son importantes y a menudo específicas para cada sistema.

En este trabajo se realizó inyección de fallas basada en emulación para acelerar el análisis del sistema citado en el apartado anterior [21] y sobre un sistema más complejo que reproduce el funcionamiento de los nodos embarcados en un automóvil durante una maniobra normalizada en la que se realiza un viraje en U [20].

La técnica de inyección de fallas por emulación de circuito en prototipo hardware fue introducida en forma independiente en [3] y en [1] para inyectar fallas del tipo “stuck-at”. Otros antecedentes de esta técnica incluyen trabajos en el Politecnico di Torino [6] anteriores a las pasantías realizadas para esta tesis. En esos trabajos, para inyectar fallas del tipo “bit-flip” se modifica el circuito de la siguiente manera: para cada

elemento de memoria en el cual se desea inyectar fallas (FF1) se agrega un FF adicional (FF2) en el cual se almacena un “1” si se desea perturbar el elemento de memoria asociado y un cero en caso contrario. En el instante de inyección se conecta a la entrada de FF1 el exor entre su entrada original y el valor almacenado en FF2, complementando de esa manera el valor cargado si FF2 está activado. Los FF adicionales se conectan entre ellos en una cadena de registros de desplazamiento denominada “mask chain”. Esta cadena de registros de desplazamiento se utiliza también para observar el valor almacenado en los elementos de memoria del circuito en un instante determinado.

PARTE II

PROTOCOLO CAN Y REDES A BORDO DE AUTOMÓVILES

Esta parte consta de dos capítulos. En el primero de ellos se presentan los aspectos del protocolo CAN que resultaron más importantes para el desarrollo del controlador y los experimentos de inyección de fallas.

Se presenta también un breve resumen sobre la utilización de redes de comunicaciones para interconectar las unidades electrónicas de control (ECUs) con que están equipados los automóviles en la actualidad. El objetivo perseguido en esta descripción es dar una idea de los requerimientos que se imponen sobre estos sistemas en aspectos como temporización, confiabilidad y posibilidad de consecuencias catastróficas de fallas, para ilustrar los criterios seguidos en la clasificación de malfuncionamientos en los experimentos descritos en la Parte III.

En ambos capítulos se busca además poner juntos conceptos y referencias recogidos durante los estudios realizados para la presente tesis, como forma de facilitar posibles estudios posteriores.

Capítulo 3

Protocolo CAN

El protocolo CAN (Controller Area Network) fue desarrollado inicialmente por Bosch [37] [42] a mediados de la década del 80 y posteriormente fue normalizado como el estándar ISO 11898 en 1993. Ha sido utilizado por la industria automotriz para las redes a bordo de automóviles desde 1992.

3.1. Breve descripción del protocolo CAN

Se brinda a continuación una breve descripción del protocolo. Descripciones más extensas pueden encontrarse en la propia especificación o p. ej. en [26].

CAN es un protocolo de múltiple acceso con detección de colisión no destructiva, resuelta por prioridades. La especificación define dos valores posibles para el estado del bus: recesivo y dominante. Si el bus es manejado simultáneamente por más de un nodo el valor dominante prevalece sobre el valor recesivo. Usualmente el valor recesivo está asociado al valor lógico “1” y el valor dominante al valor lógico “0”, de manera que la lógica del bus puede interpretarse como un “and” cableado.

Los datos se intercambian en tramas que contienen un identificador (ID) y entre 0 y 8 bytes de datos. El campo ID de cada trama no identifica al nodo que envía el mensaje sino al tipo de mensaje. Es responsabilidad del diseñador de cada aplicación definir como se asignan los ID a las tramas generadas por los diferentes nodos del sistema.

Las tramas con información pueden ser de dos tipos: tramas de datos o requerimientos de transmisión remota (tramas RTR). Las tramas RTR solamente tienen encabezado y se utilizan para solicitar la transmisión de una trama de datos con el mismo identificador.

Existen dos formatos posibles para el encabezado de las tramas, que difieren en el campo ID y en algunos campos auxiliares de control. En el formato estándar el campo ID es de 11 bits. Este formato era el único disponible en la primera versión del protocolo. En el formato extendido en cambio, el campo ID está formado por la

concatenación del ID básico de 11 bits con el ID extendido de 18 bits, totalizando 29 bits.

Si se produce una colisión entre dos nodos que inician la transmisión de una trama al mismo tiempo, se utiliza un mecanismo de arbitración por prioridad para resolver el conflicto. Lo que se hace es ir comparando cada bit a medida que los contendientes escriben sobre el bus el campo ID y algunos bits adicionales del encabezado de la trama (IDE que indica el formato de ID y RTR que indica el tipo de trama), que conjuntamente se denominan campo de arbitración. A la primera diferencia en estos campos, el nodo que escribió un valor recesivo lo detecta porque lee del bus un valor dominante y debe retirarse de la contienda convirtiéndose en receptor. El nodo que puso el valor dominante sigue adelante con la transmisión sin notar siquiera que hubo un conflicto. El nodo perdedor podrá reintentar una vez finalizada la transmisión de la trama por el ganador.

El protocolo CAN fue desarrollado para ser utilizado en sistemas con fuertes requerimientos de confiabilidad, por lo que fue provisto con múltiples mecanismos de detección de errores:

- error de bit. Cada nodo cuando escribe sobre el bus verifica que se lea el mismo valor escrito. Existen un par de excepciones a esta regla (arbitración de prioridades, reconocimiento de recepción de trama).

- error de bit de relleno (stuff). el transmisor inserta bits de relleno para evitar secuencias de más de 5 bits iguales. Es un error si se reciben 6 bits seguidos iguales.

- error de CRC. El transmisor agrega al final de la trama un código de redundancia cíclica de 15 bits calculado sobre la trama completa incluyendo el encabezado.

- error de forma y error de reconocimiento. Se detectan cuando se recibe alterado el valor de algunos bits de formato fijo en la trama.

Un análisis detallado de los mecanismos de detección de errores de CAN puede encontrarse en [28].

Para procurar asegurar la consistencia de la información recibida por todos los nodos, si un nodo detecta un error debe escribir sobre el bus un valor dominante durante seis períodos de bit, forzando de esta forma un error de bit de relleno en el resto de los nodos conectados al bus. De esta manera la transmisión no se completa y el nodo transmisor deberá retrasmitir la trama. A esa secuencia de seis bits en valor dominante se le llama trama de error. Como se muestra más adelante, ese objetivo de asegurar la consistencia en la recepción de las tramas no siempre puede lograrse.

Existe un mecanismo de confinamiento de fallas para evitar que el bus se bloquee completamente en presencia de una falla permanente. Para eso cada nodo lleva la cuenta de la cantidad de errores recientes de transmisión y de recepción. En funcionamiento normal, un nodo se encuentra en el estado “activo frente a errores” (“error active”) y su comportamiento frente a un error es el descrito líneas arriba. Si el valor de los contadores de errores supera un umbral predeterminado, entonces el nodo pasa al estado “pasivo frente a errores” (“error passive”) en el cual el nodo puede seguir enviando y transmitiendo tramas pero se debe auto imponer las siguientes restricciones: a) luego de transmitir una trama debe esperar ocho tiempos de bit adicionales antes de intentar transmitir nuevamente, para permitir iniciar una transmisión a otros nodos; b) al detectar un error debe generar una trama de error especial sin escribir el valor dominante sobre el bus. Si se siguen detectando errores y los contadores superan un segundo umbral, entonces el nodo pasa al estado “bus off” en el cual no debe manejar el bus por ningún motivo. Se debe mantener en este estado hasta que observe el bus inactivo por un período prolongado preestablecido.

3.2. Protocolos de capas superiores sobre CAN

Los sistemas distribuidos que utilizan un bus CAN en general pueden modelarse en un modelo simplificado de tres capas: física, enlace y aplicación. La especificación de CAN publicada por Bosch [42] se refiere solamente a las subcapas inferiores de la capa de enlace. Para la capa física existen varias alternativas en uso, siendo la más difundida la especificada en la norma ISO 11898-2 [51].

En el presente trabajo no fue utilizado ninguno de los protocolos existentes para la capa de aplicación. Se trabajó directamente sobre la especificación CAN de Bosch, se desarrolló un controlador CAN que la implementa y se ensayaron los mecanismos de

inyección de fallas en aplicaciones sencillas apoyadas directamente sobre los servicios provistos por el controlador CAN desarrollado.

Si bien no se profundizó en las soluciones en uso para implementar las capas superiores del modelo de referencia OSI, se listan a continuación algunas de las que aparecen referidas más a menudo.

En un inicio la mayoría de las soluciones fueron propietarias: cada fabricante implementó sus propios protocolos y reglas para solucionar problemas como el mapeo de las señales de la aplicación a las tramas CAN, el manejo de señales periódicas, la asignación de IDs a cada tipo de mensaje y la definición de primitivas de comunicación. Rápidamente aparecieron y se difundieron varios protocolos de uso más abierto.

Un ejemplo de estos protocolos para aplicaciones automotrices es el conjunto de estándares desarrollado por el consorcio OSEK/VDX [48] formado por fabricantes europeos de automóviles y partes, en particular el protocolo OSEK Communication [32]. En realidad OSEK Communication es un concepto mucho más amplio en que CAN es una de las posibles redes de comunicación.

Varios protocolos de capas superiores se utilizan en aplicaciones industriales. Uno de ellos es CANopen, desarrollado y gestionado por la organización CAN in Automation (CiA) [43]. CANopen define en forma estandarizada el mapeo de datos de la aplicación a varios tipos de “Data Objects”, que luego envía como tramas CAN. Además de CANopen, en el sitio web de CiA se listan otros protocolos utilizados en aplicaciones industriales (CANKingdom [44], DeviceNet [47]).

3.3. Dependabilidad y CAN

El protocolo CAN es un protocolo reconocido por sus buenas características desde el punto de vista de su robustez frente a fallas, y desde este punto de vista ha sido exhaustivamente estudiado por la academia y la industria. Esto ha permitido evaluar su influencia en la *dependabilidad*. Ha permitido también detectar varias debilidades que no pueden ser resueltas por CAN, y esto ha estimulado a buscar soluciones agregando funcionalidades en las capas superiores o desarrollando nuevos protocolos que probablemente desplacen a CAN a mediano plazo para algunas aplicaciones.

La mayoría de los trabajos sobre CAN se concentran en estudiar el efecto de errores en la comunicación: uno o varios bits en la trama invierten su valor. Las causas más estudiadas para estos errores son fallas a nivel del cableado del bus o del driver que fuerza un valor sobre el bus. Esto es totalmente razonable porque en un ambiente agresivo, sujeto a golpes y vibraciones como es un automóvil es esperable que por lejos las fallas de cableado sean las más frecuentes. Las herramientas desarrolladas en el presente trabajo permiten no solo inyectar fallas a nivel de capa física sino también a nivel de la lógica del controlador de acceso al bus. Si bien estas fallas son menos frecuentes sus efectos pueden también ser muy graves como se muestra en Parte III.

Muchos de los estudios y mejoras propuestas pueden agruparse como esquemas de cableado redundante para soportar cortocircuitos o falsos contactos en el bus. Algunas soluciones propuestas involucran replicar tanto el cableado físico como el controlador CAN, quedando por cuenta de la aplicación o de un árbitro decidir cuál es la réplica que falla en caso de diferencias. Una solución propuesta por Rufino [51] aprovecha el comportamiento de los niveles recesivo y dominante de CAN. Básicamente lo que propone es replicar solamente el medio físico y los drivers, y combinar en recepción con una compuerta AND los niveles recibidos. Es necesario agregar un temporizador y lógica adicional para detectar las fallas del tipo “stuck-at-dominant”.

El protocolo CAN tiene algunos problemas que inicialmente no fueron advertidos. El mecanismo de forzar errores de relleno mediante tramas de error explicado en el apartado anterior procura garantizar que una trama es recibida en forma consistente por todos los nodos: o todos la reciben bien o se fuerza un error en todos los nodos para que la trama sea retransmitida. Este objetivo no se cumple si algún nodo recibe un bit erróneo en una posición precisa en el penúltimo bit del delimitador de fin de trama [25]. En ese caso se dan situaciones en que la trama no es recibida en forma consistente, ya sea por duplicación (algunos nodos reciben la trama dos veces) u omisión (algunos nodos no reciben la trama). Se dice entonces que CAN no es capaz de garantizar la propiedad de difusión atómica (“*atomic broadcast*”) por sí solo, haciendo necesario agregar mecanismos para eso en capas superiores de protocolo. Un fenómeno interesante detectado en los experimentos de inyección de fallas realizados en el presente trabajo es que fallas del tipo “bit-flip” en registros del bloque

de control del controlador CAN provocaron el mismo efecto descrito más arriba [ver Parte III7.1].

Otro conjunto de debilidades reportadas del protocolo CAN tienen que ver con los mecanismos de confinamiento de fallas, pensado para identificar y aislar fallas permanentes. Navet y Gaujal muestran a través de un análisis de cadenas de Markov que en condiciones desfavorables pero realistas de interferencia en el bus es relativamente fácil entrar en el estado bus-off [52]. En otro trabajo [4] Navet sostiene que la principal debilidad desde este punto de vista es que los nodos deben diagnosticarse a sí mismos, y esto es imposible por ejemplo en caso de una falla en el oscilador del controlador CAN. El mismo argumento vale para fallas como las analizadas en el presente trabajo, que afecten al bloque de control del controlador CAN.

Otro grupo importante de trabajos y estudios relacionados con el protocolo CAN son los realizados desde la óptica de los sistemas distribuidos de tiempo real. Los trabajos en esta área estudian propiedades como la probabilidad de cumplimiento de *deadlines* y el establecimiento de cotas a los tiempos máximos de recepción de una trama en presencia de fallas; y el efecto sobre estas propiedades de la política de programación (scheduling) de los mensajes sobre el bus [7][27][29].

Desde este punto de vista es importantísimo el resultado señalado líneas arriba sobre la imposibilidad de garantizar “atomic broadcast”. Han sido propuestos varias modificaciones a CAN y nuevos protocolos que procuran corregir estas debilidades. Varios de estos protocolos sustituyen el arbitraje por prioridad utilizado por CAN para resolver las colisiones en el acceso al bus por una política de evitar las colisiones a través de esquemas “Time Division Multiple Access” (TDMA). Estos protocolos se adaptan muy bien al intercambio de mensajes periódicos (que son la mayoría en aplicaciones automotrices) ya que el acceso al bus por cada nodo se realiza cíclicamente en slots de tiempo prefijados estáticamente en el diseño del sistema, simplificando en gran forma el cálculo de cotas para los tiempos de respuesta. A estos sistemas se les llama “Time-Triggered”, mientras que a las redes como las del protocolo CAN clásico se les denomina “Event-Triggered”. Varios de los nuevos protocolos propuestos son soluciones mixtas, que reservan una ventana de tiempo dentro del ciclo TDMA para el intercambio de mensajes no periódicos.

En la Time Triggered Architecture se desarrollaron algunos de los primeros protocolos de este tipo propuestos: Time Triggered Protocol en sus variantes de alta (TTP/C) [34] y baja (TTP/A) [35] velocidad. Estos protocolos fueron desarrollados por el Grupo de Sistemas de Tiempo Real de la Universidad de Tecnología de Viena liderado por Hermann Kopetz [50], e impulsados comercialmente por la empresa TTTech [49].

Bosch desarrolló una variante de CAN, denominada TTCAN (por Time-Triggered CAN) que tiene la virtud de ser relativamente compatible con CAN [36].

Finalmente, el protocolo FlexRay Communications Systems [31] es el más firme candidato a ser utilizado en aplicaciones automotrices con muy fuertes requerimientos de tolerancia a fallas [4]. Está siendo desarrollado por un consorcio de actores de la industria automotriz europeos y estadounidenses, incluyendo fabricantes de automóviles (BMW, DaimlerChrysler, GM, Volkswagen), de partes (Bosch) y de componentes electrónicos (Philips).

Capítulo 4

Uso de redes a bordo de automóviles

Se presenta en este capítulo un muy breve resumen sobre la utilización de redes de comunicaciones para interconectar las unidades electrónicas de control (ECUs) distribuidas a bordo de los automóviles modernos. El objetivo perseguido en esta descripción es dar una idea de los requerimientos que se imponen sobre estos sistemas en aspectos como temporización, confiabilidad y posibilidad de consecuencias catastróficas de fallas, para ilustrar los criterios seguidos en la clasificación de malfuncionamientos en los experimentos descritos en Parte III. Se busca también poner juntos conceptos y referencias recogidos durante los estudios realizados para la presente tesis, como forma de facilitar posibles estudios posteriores.

Se advierte sin embargo que no es un tema en el que se haya incursionado a fondo. Para una revisión más extensa y muy actualizada de esta temática el lector puede referirse a un trabajo de Navet publicado recientemente [4].

4.1. Sistemas electrónicos a bordo

Una tendencia sostenida en los últimos años ha llevado a que los automóviles actuales estén equipados con una cantidad importante de sistemas electrónicos con finalidades que van desde mejorar la seguridad de pasajeros y conductor, hasta asistir al conductor en determinadas maniobras o mejorar el confort a bordo.

Gradualmente más y más funcionalidades van pasando a ser realizadas por las denominadas Unidades Electrónicas de Control (ECUs) formadas por sensores y actuadores manejados por un microcontrolador. Inicialmente cada ECU era autónoma pero rápidamente se vio la ventaja de comunicar ECUs con funciones diferentes para poder compartir sensores para las señales compartidas, o de partir una función en ECUs diferentes para alivianar el cableado a sensores y actuadores. Esa red de comunicación entre ECUs rápidamente evolucionó de una topología en malla a una topología en uno o varios buses al aumentar la cantidad de ECUs utilizadas en un automóvil.

Según un trabajo citado en [4] algunos automóviles de lujo actuales llegan a tener unas 70 ECUs que intercambian unas 2500 señales diferentes. El modelo de automóvil en el cual se inspiraron las aplicaciones de ejemplo elaboradas para los experimentos de Parte III7.2 cuenta con 25 ECUs distribuidas sobre 2 buses CAN.

El aumento de la presencia de sistemas electrónicos ha sido impulsado por un lado por el abaratamiento y mejora en la confiabilidad de los mismos, y por otro lado en la demanda de nuevas funcionalidades. Desde el punto de vista de los requerimientos de confiabilidad y seguridad es importante la utilización de sistemas de control para asistir al conductor en el manejo, en aspectos como el frenado (p. ej. en los sistemas Anti-lock Braking System ABS), la suspensión y hasta el control del motor. Para poder realizar esas funciones de asistencia en algunos casos se está pasando de comandar a los actuadores por medios mecánicos o hidráulicos a hacerlo por medio de mensajes enviados entre ECUs a través de la red de comunicaciones, incluso con señales críticas para la seguridad del vehículo. Un ejemplo de esto es la supresión de la barra de dirección en los sistemas denominados *steer-by-wire*. Es evidente que el uso de sistemas *X-by-wire* imponen sobre la red de comunicaciones requerimientos muy fuertes de *dependabilidad*.

4.2. Diversidad de requerimientos

Los diversos tipos de funciones para las que se utilizan subsistemas electrónicos a bordo imponen diferentes requerimientos sobre la seguridad, performance, tolerancia a fallas y características temporales de la red de comunicaciones.

Algunas de estos dominios funcionales tienen fuertes requerimientos de tiempo real. Los dominios funcionales de *powertrain* (control del motor y la transmisión) y de *chassis* (control de frenado, viraje y suspensión) son en general sistemas de control en lazo cerrado, con períodos de muestreo del orden de los 10ms. Requieren por lo tanto de la red de comunicaciones tiempos de respuesta rápidos y acotados en forma determinística y del microcontrolador una potencia de cálculo apreciable. Tienen además, en especial las funciones de *chassis*, fuertes requerimientos de *dependabilidad* ya que afectan fuertemente la seguridad de los ocupantes del vehículo. En este dominio funcional la red más utilizada en la actualidad es CAN, pero la tendencia es hacia protocolos del tipo Time-Triggered descritos en el capítulo anterior.

Otro conjunto de funciones con requerimientos totalmente diferentes son las denominadas de *carrocería* (*body)*, que agrupa a los comandos e indicadores del tablero de instrumentos y a funciones como el control de aire acondicionado y el control de posición de asientos, luces y espejos. En este caso se intercambia una cantidad grande de señales, algunas de ellas periódicas (p. ej. la velocidad del motor para ser desplegada en el tablero de instrumentos) y otras esporádicas (p. ej. el comando de accionar el alza cristales o encender el limpiaparabrisas). Los tiempos de muestreo son en general menores, y también es mucho menor la criticidad de un error y por tanto los requerimientos de tolerancia a fallas. Debido a la gran cantidad de elementos sencillos involucrados, una arquitectura habitual para la red del dominio de "carrocería" es una jerarquía en dos niveles: un backbone CAN interconecta a un conjunto de nodos, y cada nodo maneja a un conjunto de sensores y actuadores cercanos a través de algún bus serial de menor costo. El bus CAN del backbone suele ser de menor velocidad que el utilizado para *powertrain* y *chassis*. El bus serial de bajo costo más utilizado es LIN [45], desarrollado por un consorcio de fabricantes de automóviles.

Otro dominio funcional en desarrollo agrupa aplicaciones de entretenimiento, comunicación a distancia, navegación, e interfaces de usuario. Aparecen aquí requerimientos de ancho de banda y calidad de servicio para multimedia.

Finalmente el dominio denominado "*active and passive safety*" por Navet en [4] agrupa funciones que velan por la seguridad del vehículo en procura de evitar accidentes o paliar sus consecuencias (airbag, tensión de cinto de seguridad, monitoreo de presión de neumáticos, etc.).

4.3. Clasificación de redes según SAE

La clasificación introducida por la Society for Automotive Engineers (SAE) procura poner algo de orden en esta diversidad de requerimientos y soluciones para la red de comunicación. SAE define tres clases diferentes de redes según la velocidad de transmisión y el tipo de aplicaciones:

Las redes *clase A* son redes de bajo costo y baja velocidad (menor a 10kbps). Se utilizan en el dominio de carrocería para comunicar un nodo con sus sensores y actuadores. Ejemplos: LIN [45], TTP/A [35].

Las redes *clase B* (entre 10kbps y 125kbps) se utilizan como backbone en el dominio de carrocería. Ejemplos: J1850 y CAN de baja velocidad.

Las redes *clase C* (entre 125Kbps y 1Mbps) se utilizan para las aplicaciones tiempo real de los dominios powertrain y chassis. Ejemplo predominante: CAN de alta velocidad.

Si bien no está definido en la clasificación de SAE, a menudo se les llama redes clase D a las redes con velocidades mayores a 1Mbps. Ejemplos en esta clase son MOST para aplicaciones multimedia, y FlexRay [31] y TTP/C [34] para aplicaciones como *X-by-wire* con fuertes requerimientos de tiempos de respuesta y tolerancia a fallas.

4.4. Ejemplo

A modo de ejemplo se resumen las características de la red de comunicaciones en la cual están inspiradas las aplicaciones de prueba del apartado Parte III7.2., que corresponde a un automóvil de lujo del año 2001.

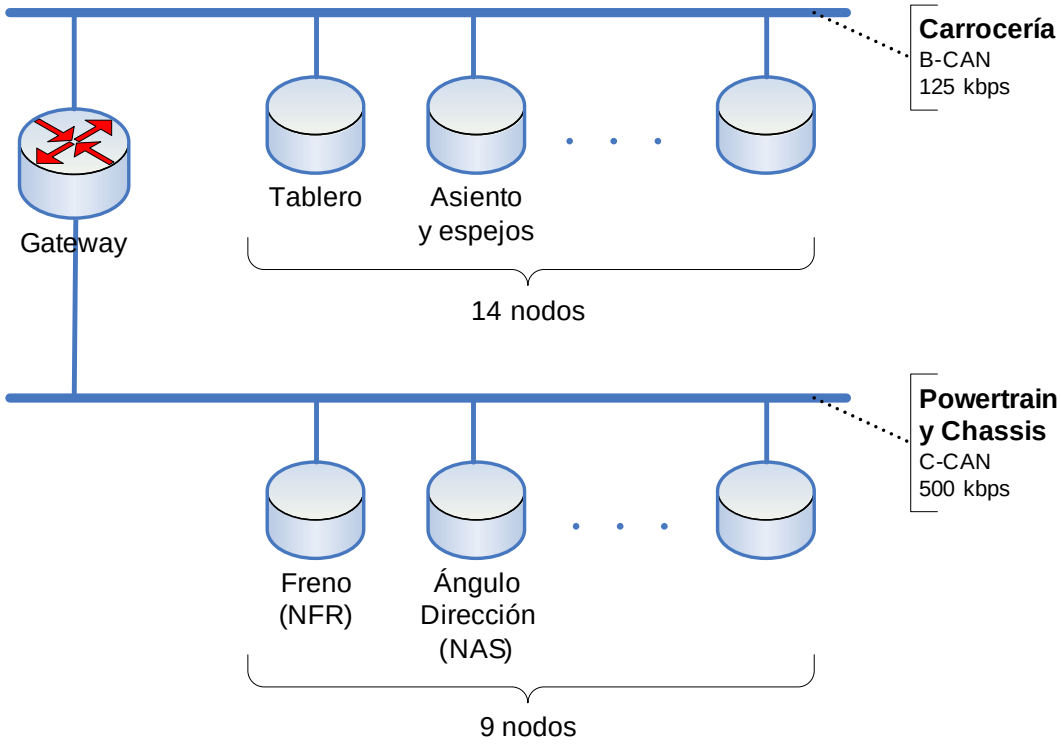

Fig. 4 Redes clase B y clase C en vehículo ejemplo.

En ese vehículo los dominios *powertrain* y *chassis* están soportados por nueve nodos comunicados por una red CAN de alta velocidad (C-CAN de 500 Kbps), más un

décimo nodo que hace las veces de gateway con el bus del dominio *carrocería*. Este segundo bus es también una red CAN pero de menor velocidad (B-CAN de 125 Kbps) que interconecta a ese gateway con catorce nodos adicionales. Cada uno de estos nodos controla a varios sensores y actuadores, p. ej. uno maneja el tablero de instrumentos, otro controla la posición del asiento del conductor y de los espejos.

El tráfico en C-CAN corresponde predominantemente a señales periódicas con períodos de muestreo de 5 ms o 10 ms. En cambio la mayoría de las señales intercambiadas en el bus B-CAN son esporádicas o periódicas con períodos del orden de un segundo.

El tráfico utilizado en los experimentos de inyección de fallas presentados en el capítulo Parte III7.2 corresponden al tráfico de tiempo real del bus C-CAN. Se consideraron todos los nodos involucrados en la maniobra de viraje analizada, que son seis en total. En ausencia de retransmisiones la utilización total del bus es de alrededor de 27%, mientras que los seis nodos incluidos en el modelo ocupan por sí solos el 19% de la capacidad del bus.

PARTE III

DESARROLLOS Y EXPERIMENTOS

Capítulo 5

Desarrollo del controlador de protocolo CAN

En este capítulo se describe el modelo VHDL del controlador de acceso al bus CAN que fue desarrollado para realizar las campañas de inyección de fallas. Se presentan las motivaciones que llevaron a desarrollarlo, una descripción de la arquitectura del controlador y el estado actual del desarrollo.

5.1. Requerimientos. ¿Por qué un nuevo controlador?

Desde el inicio, uno de los objetivos de este trabajo fue el de analizar no solamente el efecto de fallas a nivel de cableado en el bus CAN, sino principalmente el efecto de fallas sobre los registros internos de los controladores de acceso a la red. Para eso el controlador que se utilice no puede ser un modelo de caja negra, sino que por el contrario se necesita tener acceso al código fuente del mismo para realizar las modificaciones necesarias para inyectar las fallas.

Se decidió desarrollar el controlador en lenguaje VHDL. Como además se pretendía realizar inyección de fallas basada en emulación en un prototipo hardware, el desarrollo no pudo limitarse a un modelo de simulación sino que debió ser totalmente sintetizable.

Como requerimiento adicional, dado que las plataformas utilizadas en las dos instituciones involucradas son de diferentes fabricantes (Altera en el IIE, Xilinx en POLITO) el código resultante debía ser portable y sintetizarse para ambos ambientes de desarrollo. Por este motivo el desarrollo se hizo en VHDL básico, sin utilizar elementos de las bibliotecas provistas por los fabricantes (LPM, Logicore).

A la fecha de inicio del presente trabajo (enero 2003) no se encontró ningún controlador CAN sintetizable con código fuente abierto, por lo que se inició el desarrollo de un controlador completamente nuevo. Se aprovecharon experiencias y resultados de un proyecto anterior realizado por estudiantes del curso Diseño Lógico 2 bajo mi dirección [39].

Más adelante fue liberado un controlador para protocolo CAN con código abierto [38]. Este controlador fue desarrollado inicialmente en lenguaje Verilog y algunos meses más tarde fue lanzada una versión VHDL. Este controlador no fue utilizado en el presente trabajo porque cuando fue publicado ya se encontraba muy avanzado el desarrollo de nuestro propio controlador.

El controlador desarrollado cumple con la especificación CAN 2.0B [42] a excepción de los mecanismos de confinamiento de fallas que aún no han sido implementados.

5.2. Descripción del controlador y estado actual del desarrollo

El bloque principal del controlador es el bloque *can_interface*, que implementa el controlador para un nodo de la red. Este bloque se conecta a través de las señales *tx_data* y *rx_data* con el bus CAN, y a través de los buses *tx_msg*, *rx_msg* y señales de handshake con la aplicación que lo utiliza.

Un componente sintetizable adicional modela el comportamiento del bus CAN, que esencialmente consiste de una compuerta AND con tantas entradas como nodos conectados.

Finalmente se diseñó un modelo de red genérica (*can_system*) compuesto por N controladores que se comunican a través del bus CAN. La cantidad de controladores N es un parámetro que puede ser fijado por el usuario final del modelo en el momento de instanciar un *can_system*.

La primera versión del controlador fue desarrollada durante la primera pasantía en Torino a comienzos de 2003 y fue utilizado para una primera serie de experimentos de inyección de fallas a nivel del bus y de algunos registros internos. Durante la segunda pasantía en 2004 se modificó la lógica de *saboteurs* y *probes* para permitir observar y controlar los elementos de memoria internos y se extendió esta posibilidad a la totalidad de los registros, incluyendo los FF que almacenan el estado de las máquinas de estado.

A la fecha no ha sido implementado aún el mecanismo de confinamiento de fallas previsto por CAN. Esto significa que el controlador siempre funciona en el estado *error active* definido en el protocolo. Se estima que esto no tuvo una influencia mayor

en los resultados por dos motivos: a) por un lado los mecanismos de confinamiento de fallas de CAN apuntan a aislar fallas permanentes y en los experimentos realizados se inyectaron fallas del tipo bit flip o del tipo stuck-at de corta duración. b) como fue señalado en [Parte II3.1], los mecanismos de confinamiento de fallas de CAN son de poca utilidad si lo que está fallando es el propio controlador CAN.

Por ser completamente sintetizable, el controlador CAN desarrollado pudo ser utilizado en campañas de inyección de fallas tanto por simulación como por emulación en prototipo hardware.

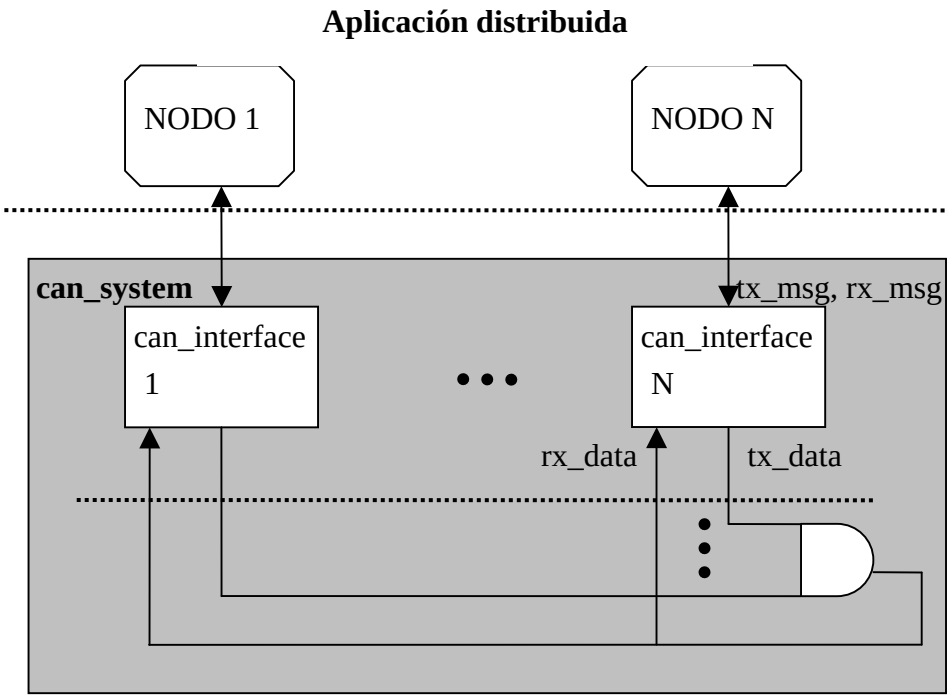

Fig. 5 Sistema CAN con varios nodos.

Capítulo 6

Ambiente para inyección de fallas

Se desarrolló un ambiente para la realización de campañas de inyección de fallas a nivel de los controladores de accesos al bus CAN a efectos de analizar cómo afectan a una aplicación distribuida en varios nodos. La arquitectura de este ambiente está compuesta por dos módulos.

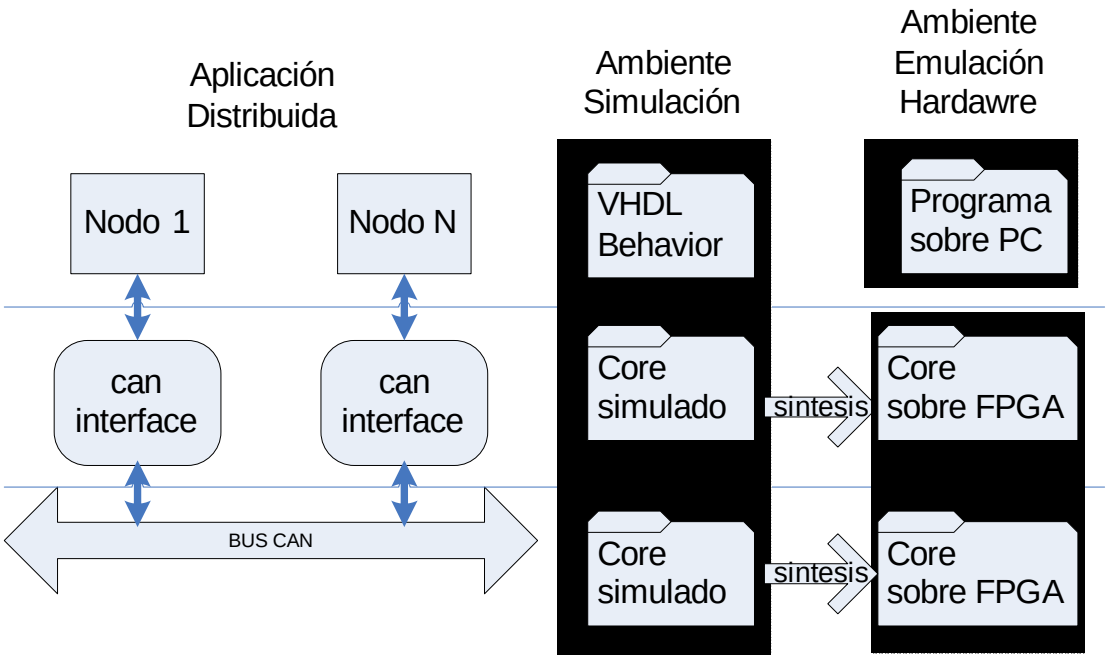

Fig. 6 Despliegue del sistema en los dos ambientes desarrollados.

El primer módulo abarca las capas más bajas incluyendo el bus CAN y los controladores de acceso al bus para cada nodo. Este módulo está basado en el modelo VHDL descrito en el apartado anterior, con el agregado de elementos adicionales para soportar la inyección de las fallas. El módulo completo sigue siendo sintetizable por lo que el mismo modelo se utiliza indistintamente para campañas basadas en simulación o en emulación sobre prototipo hardware.

El segundo módulo en cambio debe encargarse de dos tareas: por un lado emular el comportamiento de las aplicaciones que corren en cada uno de los nodos de la red; y por otro lado gestionar la inyección de las fallas y el registro de los resultados. A

diferencia del modelo de la red, la emulación de la aplicación sufrió fuertes cambios de una a otra implementación, manteniendo solamente la filosofía general.

El conjunto de herramientas desarrollado permite inyectar fallas transitorias o permanentes a nivel del bus CAN, y fallas del tipo bit-flip en los registros internos del controlador de acceso al bus. El énfasis del trabajo fue puesto en el segundo tipo de fallas ya que el efecto de las fallas permanentes a nivel del bus CAN ha sido extensamente estudiado con anterioridad [51].

Se describe a continuación las modificaciones realizadas al modelo de la red para soportar la inyección de fallas, para luego describir el ambiente basado en simulación y el ambiente basado en emulación en prototipo hardware.

6.1. Soporte para inyección de fallas

En los trabajos realizados en el marco de esta tesis se ensayaron varias arquitecturas para la inyección de fallas y la observación de señales sobre el modelo descrito en el apartado anterior.

Si bien el énfasis fue puesto en la inyección de fallas en el interior del controlador de acceso al bus, también se inyectaron fallas sobre el propio bus CAN.

Se describe a continuación las diferentes estructuras agregadas al modelo del bus CAN para soportar la inyección de fallas.

Inyección a nivel del bus CAN

Para modelar las fallas a nivel del bus se insertó entre el bus y la entrada *rx_data* del controlador un *saboteur* que permite perturbar la entrada proveniente del bus de acuerdo a lo comandado por la entrada *interference*.

Como se puede ver en el código VHDL del recuadro, este módulo permite complementar, forzar a nivel bajo o forzar a nivel alto el valor de la entrada. El primer tipo de perturbación corresponde a fallas transitorias del tipo bit-flip, provocadas habitualmente por radiación o interferencias. Las restantes perturbaciones permiten inyectar fallas del tipo "*stuck-at*", que podrían ser causadas por daño permanente o temporal a nivel de capa física.

Tanto el tipo de falla inyectada como la duración de la misma se controlan manejando la entrada *interference*.

```
architecture Behavioral of bus_interface is
begin
with interference select
      dout <=     not din     when interference_error,
                  '1'         when interference_stuck_at_recessive,
                  '0'         when interference_stuck_at_dominant,
                  din         when others;
end Behavioral;
```

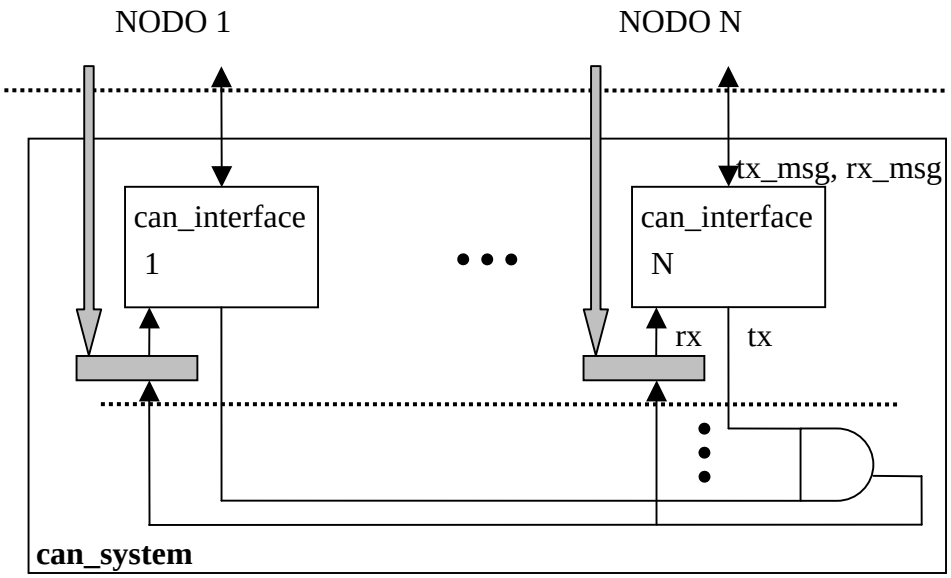

Fig. 7 can_system con soporte para inyección de fallas a nivel del bus.

Inyección en el interior del controlador CAN

En el interior del controlador se adoptó el modelo de fallas del tipo bit-flip, en el cual se invierte el contenido de un elemento de memoria en el instante de inyección. Para implementarlo fue necesario modificar y agregar elementos al circuito interno y dotar al sistema de una interfaz adicional para comandar la carga y lectura de los registros y la inyección de las fallas.

Primera versión

En los primeros experimentos esto se realizó modificando el código VHDL en nivel RTL en el modelo original, en particular las sentencias donde se realiza la carga de los registros que fueron seleccionados para la inyección de fallas. Una señal de control permite poner al circuito en modo normal o modo inyección. En el modo inyección se inhibe la transferencia de datos en todos los registros excepto aquellos en los que está habilitada la inyección, controlando con una máscara en cual o cuales de esos registros se realiza un bit-flip.

Un ejemplo simplificado del código VHDL necesario para esto se muestra en el recuadro.

```
process(clk, reset)
begin
if (reset = '1') then
      ... – Inicialización
elsif (clk'event and clk='1') then
      if (fi_mode = '1') then
      -- Inyección de falla
            reg <= reg xor mask;
      else
      -- Asignaciones normales a todos los registros
            ...
      end if; -- fi_mode
end if; -- reset and clk
end process;
```

A menudo los FF con que vienen equipados los FPGA disponen de una entrada de habilitación para simplificar la implementación de transferencias condicionales. En el caso del controlador CAN la mayor parte de las transferencias están habilitadas por una señal *sample_point* que indica el instante de muestreo seleccionado para cada bit. Esto simplificó la inhibición de las transferencias en los módulos del circuito en los que no se desea inyectar fallas ya que en ese caso basta con inhibir la señal *sample_point* mientras se está en modo inyección.

```
n_sample_point <= sample_point and not fi_mode;
```

Segunda versión

Para la segunda serie de experimentos realizada en el año 2004 se adoptó un enfoque más sistemático, insertando saboteurs como un componente estructural en la entrada de los registros que se desea perturbar con fallas.

Esto se hizo porque en esta oportunidad los requerimientos fueron mayores. En primer lugar se extendió el soporte para la inyección de bit-flips a la totalidad de los registros del controlador. Por otra parte el tráfico utilizado para activar las fallas (el conjunto A en el modelo FARM) es el resultante de una maniobra de un automóvil con una duración de 10 segundos, y por lo tanto involucra la transferencia de varios miles de tramas CAN sobre el bus. Para evitar repetir la totalidad de la maniobra en cada experimento se decidió dotar al circuito no solamente de la capacidad de inyectar fallas, sino también de escribir y leer el valor de todos los elementos de memoria internos del circuito. Para eso se realiza una primera ejecución sin inyectar fallas, almacenando el contenido de todos los registros en instantes predeterminados denominados *checkpoints*. A esta ejecución se le denomina *golden run*. Luego, para cada falla a inyectar se puede iniciar el experimento llevando el sistema al estado registrado en el *checkpoint* anterior al instante de inyección y detener la emulación en el primer checkpoint en que el estado leído coincide con el estado almacenado en la *golden run*. Esta estrategia es la que determinó que se equipara con saboteurs a la totalidad de los registros.

Se presenta a continuación las modificaciones realizadas al modelo original para agregar estas funcionalidades.

Dado que el componente saboteur se inserta a la entrada de cada registro, la modificación del circuito se ve simplificada si en el código VHDL original la entrada del registro está evaluada explícitamente mediante un proceso o una asignación concurrente. Este es un estilo de codificación habitual en VHDL, pero si el código original no estuviera escrito de esta manera deberá ser modificado para llevarlo a este estilo antes de poder insertar los saboteurs como elementos estructurales. El recuadro muestra un ejemplo de ese estilo de codificación para un registro *reg*.

```
process(clk)
begin
if (clk'event and clk='1') then
      if (enable='1') then
            reg   <=    next_reg;
      end if;
end if; -- clk
end process;
...
next_reg <= -- expr. combinatoria, próximo estado del registro
```

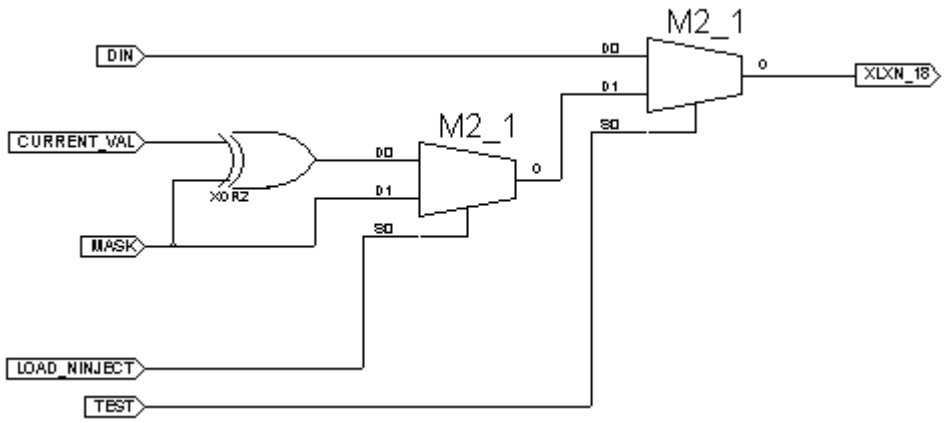

Fig. 8 Diagrama simplificado del saboteur utilizado.

La Fig. 8 muestra un diagrama simplificado del saboteur utilizado. A la salida del bloque se conecta o bien la entrada original Din durante el funcionamiento normal, o bien el EXOR entre el valor actual del registro y la máscara para introducir un bit-flip, o bien el valor de la máscara sin modificar para cargar un nuevo valor en el registro. El modo de funcionamiento se controla con las entradas TEST y LOAD_NINJECT, denominadas en conjunto FI_MODE,

Además de insertar el bloque saboteur en la entrada D de cada flip-flop, un multiplexor selecciona si la entrada de habilitación de los registros proviene de la señal generada por el circuito o de una entrada de control adicional. La Fig. 9 muestra la conexión de un bloque saboteur a un registro.

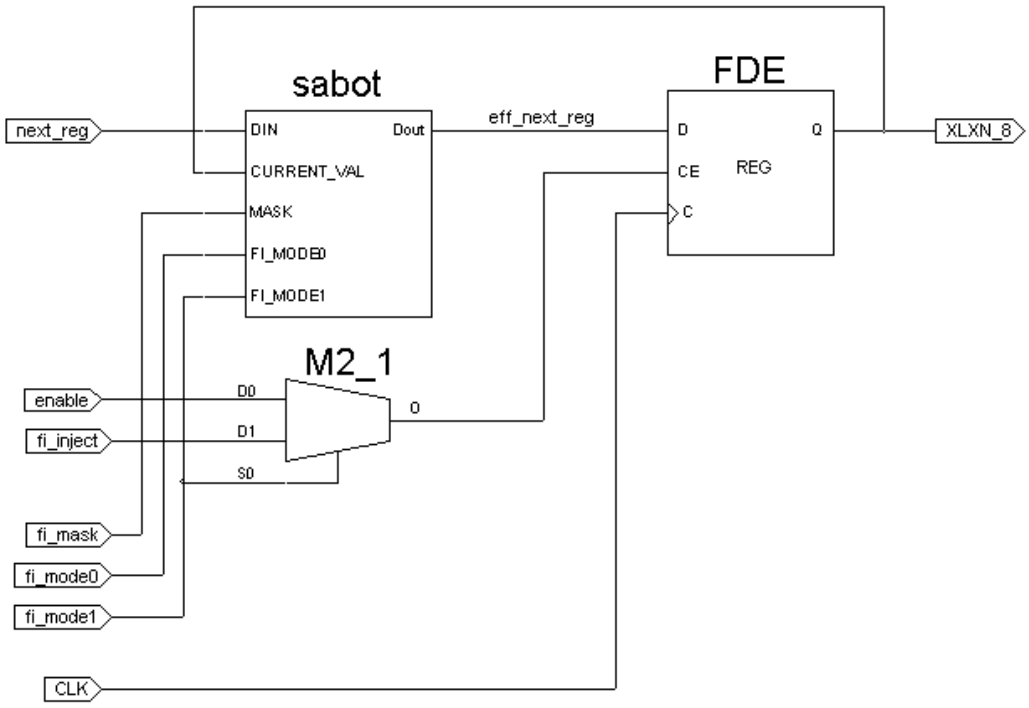

Fig. 9 Conexión del bloque saboteur

El diagrama simplificado presentado líneas arriba no muestra como se selecciona a cuál de los registros internos se le va a aplicar una operación de carga o inyección de falla. Una solución habitual para esto [6] es implementar una cadena de registros de desplazamiento (“mask-chain”) en la cual se carga la máscara en forma serial para cada uno de los registros del circuito. Si la operación a realizar es una inyección de falla el registro a perturbar se selecciona por el valor de la máscara, en una operación de carga no se puede seleccionar un registro en particular, debiéndose cargar simultáneamente todos los registros.

En nuestro caso se prefirió utilizar un bus paralelo de 32 bits y una decodificación más tradicional para cargar los registros de máscaras. Para eso se equipó a la interfaz de test con señales adicionales (fi_node_sel y fi_sel) para seleccionar cuál es el registro a perturbar y se agregó al componente saboteur un parámetro (un GENERIC de VHDL) que identifica al registro asociado a cada instancia.

Para implementar los *probes* a efectos de observar el contenido de los registros se utilizó la misma señal fi_sel para multiplexar el contenido de los registros sobre una señal de salida a la que denominamos fi_obs.

La interfaz completa que agrupa todas las señales utilizadas para la inyección de fallas en el interior del controlador finalmente quedó conformada como sigue:

```
fi_mode     : in  fi_mode_type;
fi_inject   : in  std_logic;
fi_node_sel : in  STD_LOGIC_vector (num_of_cans downto 1);
                        -- selecc. Nodo
fi_sel      : in  std_logic_vector ((SIZE_FI_SEL-1) downto 0);
                        -- selecc. registro dentro del nodo
fi_mask     : in  std_logic_vector (31 downto 0);
                        --selecc. bit dentro del registro
fi_obs      : out std_logic_vector (31 downto 0);
```

La principal ventaja de la arquitectura elegida con respecto a una cadena de registros de desplazamiento es que permite direccionar directamente el registro elegido para la inyección de falla, sin necesidad de cargar en forma serie toda la cadena de máscaras. También permite acceder directamente a un registro para leer o cargar su valor, pero esto no es tan útil ya que en los experimentos de inyección de fallas estas operaciones no se realizan en forma individual sobre un registro sino que se hacen en cada checkpoint sobre la totalidad de los elementos de memoria del circuito. Igualmente la velocidad lograda fue mayor a la solución tradicional del shift register ya que las trasferencias se hacen a través de un bus de 32 bits en lugar de en forma serial.

6.2. Ambiente basado en simulación

Para la inyección de fallas basada en simulación se utilizó el simulador Modelsim[46].

Se desarrollaron entidades VHDL para emular el comportamiento de la aplicación que corre en cada nodo y para inyectar fallas en la interfaz de bus o en el controlador CAN. A esas entidades se les denominó *transactor*, siguiendo la filosofía presentada en [40].

Para hacer una simulación del sistema completo, en un testbench se instancia un sistema CAN con la cantidad de nodos deseada. A cada nodo se le asigna un número identificador y se le conectan tres *transactor* que manejan sus entradas y salidas:

- *transactor*: Emula el comportamiento de la aplicación del nodo. Se comunica con el controlador a través de las señales tx_msg, rx_msg y señales de handshake.

- *fi_transactor*: Se encarga de inyectar fallas y cargar o leer el valor de los registros internos del controlador CAN. Para eso maneja la interfaz descrita en el apartado “Inyección en el interior del controlador CAN”. (En la segunda versión desarrollada en 2004 un solo componente fi_transactor controla la inyección de fallas para todos los nodos, utilizando la señal fi_node_sel para seleccionar el nodo en el cual se inyecta la falla.

- *bi_transactor*: Se encarga de inyectar fallas en la entrada rx_data proveniente del bus CAN de acuerdo a la descripción en “Inyección a nivel del bus CAN”.

Para los tres tipos de transactores se implementó una arquitectura que lee comandos de un archivo de texto de configuración. Para la emulación de la aplicación se implementaron además otras arquitecturas como se describe más adelante.

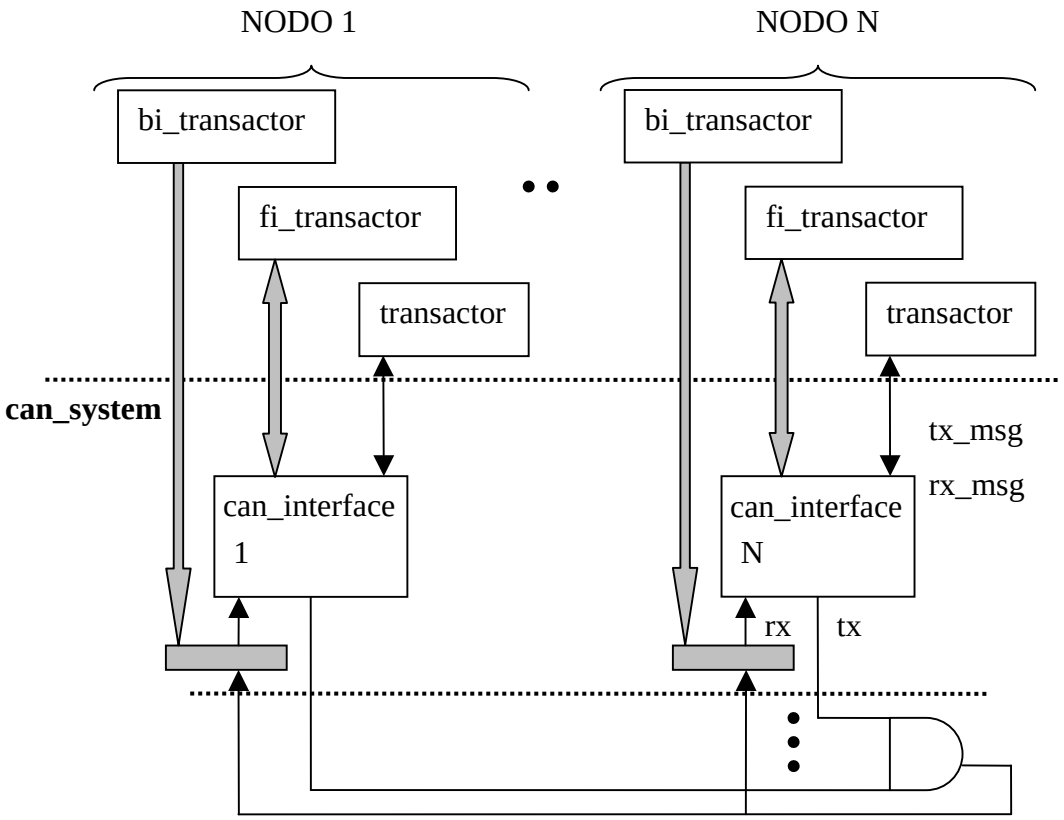

Fig. 10 Testbench para el sistema distribuido completo

Si N es la cantidad de nodos del sistema, en el testbench existen N transactors de cada tipo. Estos 3N componentes actúan en forma concurrente durante la simulación.

La sintaxis de los comandos interpretados por los transactors es la siguiente:

```
[mod_label][time][cmd][parameters...]
```

Donde

- mod_label: es un número que identifica el nodo que debe ejecutar el comando.
- time: es el instante (en ns) en que debe ejecutarse el comando
- cmd: string (8 downto 1) es el nombre del comando
- parameters: parámetros que varían en cantidad y tipo dependiendo del comando

Cada transactor procesa el archivo de texto de a una línea por vez. Los comandos deben estar ordenados por tiempo creciente. Si el primer campo no coincide con el número de nodo al cual está conectado el transactor el comando es ignorado y se pasa a la siguiente línea del archivo. Lo mismo sucede si el nombre del comando no es reconocido, situación normal ya que todos los transactors leen los comandos del mismo archivo de configuración.

Para cada transactor se implementó un repertorio de comandos sencillo para p. ej. transmitir una trama, inyectar una falla, etc.

6.3. Ambiente basado en emulación en prototipo hardware

El ambiente para inyección de fallas por emulación desarrollado en este trabajo fue presentado por primera vez en [21]. Al igual que en el caso anterior la arquitectura está compuesta por dos módulos, denominados en este caso *Castor* y *Pollux* (gemelos, hijos de Zeus según la mitología griega).

Castor emula el comportamiento de la aplicación que corre en los nodos y consiste de un conjunto de programas ejecutando en un computador personal. Se encarga además de controlar la inyección de fallas y registrar durante cada experimento las lecturas que forman el conjunto R del modelo FARM.

Pollux por su parte es un circuito sintetizado a partir del modelo de red CAN ya descrito y cargado en una placa con FPGAs insertada en un slot del computador personal.

Emulación de aplicación: Castor

Castor es un módulo software cuyo propósito es emular las aplicaciones ejecutadas en todos los nodos de la red. Las aplicaciones se describen como máquinas de estado finitas (FSMs) que evolucionan en un conjunto de estados. Las transiciones de estado ocurren en respuesta a eventos de entrada y pueden tener asociada la generación de una acción. Los eventos de entrada pueden ser: se alcanzó el instante de un evento periódico que ocurre a intervalos fijos, se recibió una trama CAN que está pronta para ser leída o se completó la transmisión de una trama CAN y por lo tanto el controlador del nodo queda libre para transmitir una nueva trama. Las acciones generadas pueden ser la transmisión de una trama o el registro de una trama recibida.

La planificación de tareas es “round-robin” y no preemptiva: cada vez que sucede un evento de entrada el planificador de tareas invoca cíclicamente a las tareas asociadas a cada uno de los nodos. Además de manejar la ejecución de las tareas en los nodos, el planificador de tareas se encarga de controlar la inyección de fallas y actualizar la cronología de la simulación.

Se han desarrollado dos versiones de *Castor*: la primera durante mi primera pasantía en Torino a comienzos de 2003 fue desarrollada principalmente por Massimo Violante y se utilizó para la serie de experimentos que se describe en el apartado *7.1Primera serie de experimentos: interrogación cíclica de nodos esclavos*. La segunda versión fue desarrollada principalmente por mí durante la segunda estadía en Torino a comienzos del 2004 y se utilizó en los experimentos que se presentan en *7.2 Segunda serie de experimentos: maniobra de viraje en* un automóvil. Como se explica más adelante, en los experimentos de 2003 cada nodo transmite por turno respondiendo a solicitudes de un nodo que oficia de master del bus. En los

experimentos realizados en 2004 en cambio se producen colisiones en el acceso al bus: un nodo que pierde el acceso al bus debe encargarse de recibir una trama de mayor prioridad cuando tiene su propia trama pendiente de ser transmitida. Por este motivo se prefirió dotar a la tarea de cada nodo de máquinas de estado independientes para transmisión y recepción.

Ambas versiones están escritas en C. El recuadro muestra un ejemplo simplificado de la función asociada a un nodo, que puede tomarse como *template*. Se utilizan dos variables estáticas para almacenar el estado de las máquinas de estado de transmisión y recepción.

La función rxFsmNext() devuelve el próximo estado de la máquina de estado de recepción. Recibe como parámetros el número de nodo y el estado actual y se comunica con Pollux para consultar las señales de handshake de recepción y para leer las tramas que se reciban.

```
void  node_1( char rst ){
      static int txState;
      static int rxState;
      if( rst )   {
            // Disable TX request
            N_TX_REQ(1) = 0x00000000;
            txState = 0;
            rxState = 0;
      }
      else{
            txState = txFsmNext( NODE1, txState );
            rxState = rxFsmNext( NODE1, rxState );
      }
}
```

```
int   rxFsmNext(int node, int pstate) {
      int   nstate;
      nstate = pstate;
      switch( pstate ) {
            case 0: // Wait for RX completed
                  if( N_RX_STATUS(node) != 0 ) {
                        // read and process received MSG
                        ...
                        nstate = 1; // MSG received
                  }
                  break;
            case 1:     // Wait for end of pulse in RX completed
                  if( N_RX_STATUS(node) == 0 )
                        nstate = 0;
                  break;
            default:
                  break;
      }
      return( nstate );
}
```

Dos precauciones deben tenerse al diseñar las máquinas de estado:

- están prohibidos los lazos dado que el planificador de tareas invoca la función de cada nodo y debe esperar que la función del nodo retorne antes de poder comenzar la ejecución del nodo siguiente.

- se debe tener presente que la función asociada a un nodo no vuelve a ser invocada hasta que no sucede un evento (cambia alguna señal de handshake en alguno de los nodos o se produce un evento periódico) por lo que no es posible generar eventos de salida en otro instante.

Emulación de la red: Pollux

Pollux es el módulo del sistema encargado de emular la red CAN. Es un módulo hardware alojado en una placa reconfigurable insertada en un slot del computador personal. En los experimentos del año 2004 se utilizó una placa Alphadata ADM-XRC, equipada con un FPGA Virtex 2000e de Xilinx.

El núcleo de Pollux (ver Fig. 11) está formado por el modelo can_system ya descrito, instanciado con la cantidad de nodos deseada y sintetizado para el chip de la placa utilizada. Los bloques y señales sombreados en la figura son los elementos agregados al modelo básico para permitir la inyección de fallas.

El bloque Network Controller se encarga de generar las señales de reloj y de manejar las señales a intercambiar con la red CAN a través de un banco de registros mapeado en el espacio de memoria del computador personal. Por ejemplo, el macro N_RX_STATUS(node) lo que hace es leer la dirección de memoria en la que está mapeada la bandera rx_completed del nodo seleccionado. La interfaz con el host es a través del bus PCI.

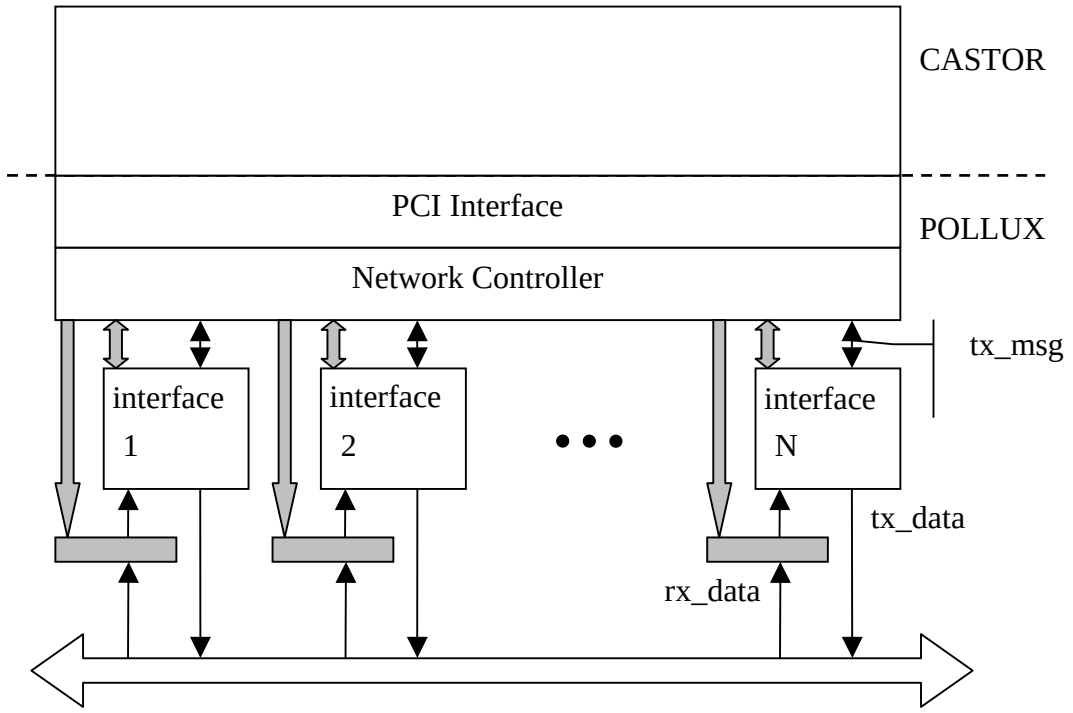

Fig. 11 Pollux: Diagrama de bloques.

Capítulo 7

Experimentos realizados

En el marco de esta tesis se realizaron varias series de experimentos para diferentes aplicaciones interconectadas sobre el bus CAN. En estos experimentos se fueron ensayando las herramientas desarrolladas para evaluar la *dependabilidad* de sistemas cada vez más complejos.

La primera serie de experimentos se realizó durante el año 2003 y en ella se utilizaron tanto técnicas de inyección por simulación como de inyección por emulación en prototipo hardware. Se tomó en ese caso como aplicación un escenario en el que un nodo que hace las veces de master interroga cíclicamente a varios nodos esclavos.

La segunda serie de experimentos se realizó a comienzos de 2004 y se trabajó sobre una aplicación tomada de la industria automotriz. En la red de control de un automóvil se modelaron los nodos involucrados en el control de freno asistido durante una maniobra de viraje. Esta maniobra está normalizada por ISO [36] y tiene una duración de 10 segundos. Dada la duración de la maniobra resultó impracticable trabajar con simulaciones, por lo que solamente se inyectaron fallas sobre el prototipo hardware. Se analizaron los modos de malfuncionamiento detectados.

Se describen a continuación con más detalle ambas series de experimentos.

7.1. Primera serie de experimentos: interrogación cíclica de nodos esclavos

Durante la primera estadía en Torino se realizaron experimentos de inyección de fallas sobre dos escenarios: en etapas tempranas del desarrollo para la depuración del diseño del controlador de acceso al bus CAN; más tarde para la evaluación de *dependabilidad* de aplicaciones distribuidas comunicadas a través de una red CAN. Se realizaron diferentes campañas para realizar estimaciones de performance del ambiente desarrollado inyectando fallas de diferentes tipos.

Para el segundo de los escenarios planteados se diseñó una aplicación procurando que fuera a la vez representativa de una aplicación real y que fuese sencilla la clasificación de las fallas.

Al diseñar un ambiente para inyección de fallas es común plantearlo en función de los conjuntos FARM (Faults, Activation, Readouts, Measures) que se describen a continuación.

El conjunto F: Fallas

En las diferentes campañas se consideraron fallas en dos niveles de la red: en el interior de los controladores de acceso al bus y en el bus CAN propiamente dicho.

Para inyectar fallas en el bus se utilizó la arquitectura descrita en la primera parte del apartado *6.1 Soporte para inyección de fallas* basada en saboteurs insertados en la entrada a cada nodo desde el bus CAN. Se inyectaron fallas del tipo bit-flip con duraciones de 2 y 10 períodos de reloj.

En el interior de los controladores de acceso al bus CAN, mientras tanto, se inyectaron fallas del tipo bit-flip en elementos de memoria. Para ello se utilizó el esquema descrito en la segunda parte del apartado *6.1 Soporte para inyección de fallas* en su primera versión. Para inyectar fallas se seleccionaron dos contadores de 7 bits que determinan la cantidad de períodos de reloj que la máquina de estado permanece en cada campo de la trama CAN. Uno de los contadores seleccionados es utilizado por la máquina de estados que controla la transmisión y el otro por la que controla la recepción de las tramas.

La falla a inyectar queda determinada entonces por el instante de inyección y la ubicación de la falla, definida esta última por el número de nodo a perturbar y la posición del bit a complementar (elegida entre los 14 flip-flops de los 2 contadores equipados para inyección de fallas). El instante de inyección se eligió al azar con distribución uniforme dentro de la duración de la transmisión de las tramas en el caso sin fallas. La ubicación también se seleccionó al azar con distribución uniforme.

Conjunto A: tráfico destinado a Activar las fallas

Las inyecciones de fallas en el primer escenario se realizaron en etapas muy tempranas del desarrollo como herramienta para la depuración y validación del diseño del core de controlador de acceso al bus CAN. Dado que el protocolo incluye mecanismos para la detección de errores de diverso tipo la prueba de esos mecanismos requiere presentar al sistema entradas con errores. Para esto se utilizaron redes con un transmisor y un receptor y se simularon transmisiones sencillas de una trama más las tramas de error y retransmisiones que resultaran. Para este fin se utilizó la inyección de fallas en el bus descrita en el apartado anterior.

Para el segundo escenario con fines de evaluación de *dependabilidad*, se diseñó un primer ejemplo sencillo de aplicación distribuida en varios nodos comunicados por una red CAN. En esta aplicación uno de los nodos actúa como master y los demás como esclavos. Cada nodo esclavo implementa un generador de secuencias (un simple contador) que es monitoreado por el nodo master.

El comportamiento del nodo master y de los nodos esclavos se describe en los recuadros que se encuentran a continuación. Luego de la inicialización el master comienza un ciclo interrogando a cada esclavo por turno, solicitando el próximo valor de la secuencia y esperando la respuesta del esclavo. Cuando recibe una trama de datos el master verifica que la respuesta recibida sea la correcta. En caso que el valor recibido difiera del valor esperado de la secuencia el efecto de la falla es clasificado en la categoría *respuesta incorrecta* definida más abajo y se aborta el experimento.

```
master(){
      for(i=0; i<NLOOPS; i++) {
            foreach slave {
                  send request(slave ID)
                  wait data frame from(slave ID)
                  compare with counter(slave ID)
                  increment counter(slave ID)
            }
      }
}
```

```
slave(){
      count = 1
      while(1) {
            wait request from(master ID)
            new data frame = count
            send data frame(master ID)
            count++
      }
}
```

En cada experimento el ciclo se repite NLOOPS veces. Al final del experimento el efecto de la falla se clasifica en las siguientes categorías:

Sin efecto: el master recibió la respuesta correcta del esclavo correspondiente en todos los casos, en los mismos instantes que para el experimento realizado sin inyectar falla.

Respuesta incorrecta: En la respuesta recibida por el master el valor recibido difiere del valor esperado para la secuencia.

Degradación de performance: luego de algunas tramas de error y retransmisiones de acuerdo con los mecanismos de detección y recuperación de errores provistos por el protocolo CAN, el sistema recupera su funcionamiento normal. La totalidad de las tramas se transmiten y reciben con los valores correctos aunque en un tiempo mayor que el esperado.

Timeout: luego de un cierto límite de tiempo prefijado, la transmisión de las tramas no logra completarse.

Campañas basadas en simulación

Primer escenario

Para el escenario de depuración del diseño del controlador se realizaron experimentos de inyección de fallas por simulación. En este caso la funcionalidad de los nodos conectados a la red y la inyección de las fallas se implementó a través de arquitecturas de las entidades transactor y bi_transactor descritas en *6.2 Ambiente basado en simulación*. En esta implementación tanto los instantes de transmisión como los instantes de inyección de la falla se determinan a través de comandos leídos desde un

archivo de texto por los bloques *transactor* introducidos en *6.2 Ambiente basado en simulación*. Los experimentos realizados resultaron una herramienta excelente para identificar y corregir fallas de diseño en la implementación de los mecanismos de detección de errores del protocolo CAN.

Segundo escenario

En la aplicación diseñada para evaluación, la funcionalidad de los nodos master y esclavos se implementó con sendas arquitecturas de la entidad *transactor*. La ejecución concurrente de los procesos de los diversos nodos que realiza el simulador VHDL permitió una implementación sencilla y elegante de la aplicación distribuida. Los resultados de estas campañas fueron inicialmente presentados en [22].

Se instanció una red de 4 nodos (un master y tres esclavos) y se repitió dos veces el ciclo de interrogación a los esclavos, totalizando 12 tramas CAN intercambiadas en cada experimento. El límite de tiempo a los efectos de los malfuncionamientos clasificados como *Timeout* se estableció de manera de permitir la transmisión de hasta unas 5 tramas adicionales.

Se realizaron en total 600 experimentos de inyección de fallas, cuyo resumen de resultados se muestra en Tabla 1.

Tabla 1: Clasificación del efecto de las fallas

Sin efecto	241
Respuesta incorrecta	4
Degradación de performance	351
Timeout	4
Total	600

Los registros elegidos para la inyección de fallas son contadores que determinan el tiempo de permanencia de la máquina de estados en el estado correspondiente a cada campo de la trama CAN. Dado que en todos los estados de la máquina de estados el contador es modificado (decrementado o inicializado para pasar al estado siguiente) es de esperar que una vez transcurrido el tiempo suficiente para que el contador llegue

a cero el nodo afectado vuelva a estar operativo. Mientras el nodo perturbado permanece erróneamente en el estado es de esperar que gracias a los mecanismos de detección de errores del protocolo CAN se detecten las violaciones producidas y se fuercen las retransmisiones que sean necesarias. El tipo de malfuncionamiento esperable para la falla inyectada era por lo tanto *degradación de performance* y esto se verificó para la mayor parte de las fallas *non silent*.

Sin embargo aparecieron algunos malfuncionamientos de los tipos *respuesta incorrecta* y *timeout*. Analizando en detalle las simulaciones para estas fallas se encuentra que en todos los casos corresponden a situaciones donde una trama no es aceptada o rechazada en forma consistente por todos los nodos. Por ejemplo, en uno de los casos el campo EndOfFrame del nodo que está transmitiendo fue prolongado por la falla inyectada, provocando que el nodo transmisor rechace la trama al detectar un error al final de su campo EndOfFrame más largo, cuando ya todos los demás nodos habían aceptado la trama como correcta. Inconsistencias de este tipo afectan el mecanismo de interrogación cíclica de la aplicación analizada, haciendo que o bien se repita una trama (con el riesgo de ser catalogada como valor incorrecto) o bien que nunca se transmita una respuesta por parte de un esclavo, bloqueando por consiguiente al master y a todo el ciclo. Por un motivo diferente, se produce el mismo fenómeno que cuando se recibe con error el penúltimo bit del delimitador de fin de trama, fenómeno señalado por Rufino en [25] y comentado en 3.3 Dependabilidad y CAN.

Las simulaciones se realizaron con el simulador Modelsim, corriendo sobre un computador Sun Enterprise 250, con reloj de 400 MHz y equipado con 2GB de RAM. La simulación de cada experimento en que se intercambian 12 tramas de 51 bits cada una insumió unos 4 segundos, o en forma equivalente en cada segundo de simulación se simula la transmisión de aproximadamente unos 150 bits sobre la red de 4 nodos.

Campañas basadas en emulación

Durante la pasantía en Torino en el año 2003 se llegaron a realizar algunos experimentos iniciales de inyección de fallas por emulación en prototipo utilizando Castor y Pollux, el ambiente descrito en [Capítulo 6.3] a efectos de depurar el sistema y estimar la performance del mismo. Posteriormente Massimo Violante realizó una campaña de inyección de fallas sobre la aplicación master-esclavos ya descrita

inyectando fallas stuck-at en el bus. El sistema desarrollado y los resultados de esta última campaña fueron presentados en [21].

La plataforma FPGA utilizada fue una placa de AlphaData [41] equipada con un chip Virtex 1000E de Xilinx y comunicada con el host a través del bus PCI.

Se instanció una red de 6 nodos, funcionando uno de ellos como master y los restantes como esclavos. La ocupación del chip para esta red fue de 21% de las LUT disponibles y 10% de los flip-flops disponibles.

En este caso el ciclo de interrogación se repitió 5 veces, por lo que en cada experimento se transmiten un total de 50 tramas CAN. Las tramas transmitidas son de 4 bytes de datos. De los resultados mostrados en Tabla 2 se deduce que la performance obtenida para la emulación es de alrededor de 100 tramas CAN por segundo. Con la carga de tráfico utilizada se pueden inyectar 2 fallas por segundo. Este resultado es más de 20 veces mejor que el obtenido en la inyección de fallas por simulación presentada en el apartado anterior.

Tabla 2: Efectos de fallas inyectadas en el bus CAN.

		Serie 1	**Serie 2**
Duración de la falla	[ciclos ck]	2	10
Sin efecto	[#]	902	187
Respuesta incorrecta	[#]	0	3
Degradación de Performance	[#]	98	808
Timeout	[#]	0	2
Total de fallas inyectadas	[#]	1000	1000
Tiempo de emulación	[seg.]	498.8	509.2

Con respecto a la clasificación del efecto de las fallas, se realizaron dos series de experimentos cambiando la duración de la perturbación introducida en el bus. En una

de las series la perturbación se mantuvo por 2 períodos y en la otra por 10 períodos de reloj. La tabla muestra un fuerte incremento en la cantidad de fallas que provocan malfuncionamiento. Esto se explica por las características del bus CAN, en que cada bit tiene una duración de varios períodos de reloj (10 períodos de reloj con los parámetros que utilizamos) pero los datos en el bus son muestreados solamente durante un período de reloj. De esta manera, cuando la duración de la falla es de solo 2 períodos de reloj es probable que no afecte a ningún instante de muestreo. En cambio en la serie en que la falla se mantiene por 10 períodos de reloj es seguro que afecta al menos el muestreo de un bit.

Resultados y conclusiones de la primera serie de experimentos

La primera serie de experimentos realizada en 2003 ya permitió extraer algunas conclusiones. En primer lugar, la performance obtenida, especialmente para las campañas basadas en emulación (más de 7000 fallas en una hora) confirmó la viabilidad de realizar campañas de inyección de fallas sobre una red con todos sus nodos modelados en detalle. Nótese que se está simulando el comportamiento del circuito completo período a período de reloj.

También se verificó que el ambiente desarrollado permite diseñar con relativa facilidad los módulos que modelan el comportamiento de la aplicación que corre en cada nodo de la red. Esto es muy útil durante la fase de desarrollo de un sistema para probar diferentes escenarios y así detectar en forma temprana falencias en el diseño.

Por último señalemos que, si bien los mecanismos de detección y recuperación contra errores previstos en el protocolo CAN funcionaron correctamente en casi todos los casos, algunas de las fallas inyectadas provocaron errores bizantinos que no fueron tratados en forma adecuada.

7.2. Segunda serie de experimentos: maniobra de viraje en un automóvil

La segunda serie de experimentos fue realizada durante mi segunda pasantía en Torino, en los primeros meses de 2004. El objetivo final de esta serie de experimentos fue poder determinar, en un análisis en dos niveles, cómo se ve afectado el comportamiento dinámico del automóvil por una falla en el controlador de acceso al bus CAN de uno de los nodos que lo controlan. Para eso se trabajó en cooperación

con otros investigadores del grupo CAD del Politecnico di Torino que disponían de un modelo dinámico del comportamiento de un automóvil [23].

Inicialmente se pensó en realizar una co-simulación, integrando ambos niveles de abstracción en un mismo experimento. Este objetivo resultó demasiado ambicioso para el tiempo de desarrollo disponible por lo que rápidamente se pasó a una arquitectura en dos fases, en que las conclusiones extraídas en la primera fase se utilizaron para modelar el espacio de fallas en los experimentos de la segunda fase.

Se describe a continuación el sistema analizado y los experimentos realizados.

Descripción del sistema

En cooperación con otros investigadores del grupo CAD del Politecnico di Torino (Mattia Ramasso, Simonluca Tosato, Fulvio Corno) se buscó establecer un ambiente para el análisis de *dependabilidad* de un sistema complejo como es un vehículo con sus sensores y actuadores comunicados por una red embarcada. La arquitectura del ambiente desarrollado está basada en dos vistas complementarias del sistema bajo análisis:

- *Modelo dinámico del vehículo*: describe el comportamiento del sistema bajo análisis como un modelo Matlab/Simulink. En esta vista se describe al sistema con un nivel de abstracción alto, pero se pierden detalles por ejemplo del funcionamiento de la red embarcada. Este modelo [23] fue desarrollado previamente por el grupo del profesor Velardocchia en el departamento de mecánica del Politecnico di Torino, en cooperación con FIAT y las simulaciones y experimentos sobre el mismo fueron realizadas por Simonluca Tosato y Mattia Ramasso, estudiantes de doctorado del grupo CAD.

- *Modelo detallado de la red*: describe en detalle algunos componentes del sistema seleccionados por el analista. En el ejemplo que se describe aquí esta vista modela en forma detallada y precisa el comportamiento y la arquitectura de la red a bordo del vehículo basada en el protocolo CAN. Para esto se utilizó el ambiente desarrollado en el marco de esta tesis descrito en [6.3].

Ambos modelos tienen extensiones que permiten realizar inyección de fallas. Al disponer de un ambiente de este tipo un diseñador puede analizar el efecto de las

fallas sobre el sistema en los dos niveles de abstracción diferentes. Utilizando el modelo dinámico del vehículo se puede hacer un análisis "*grano grueso*" del sistema para ver cómo las fallas en la red de comunicaciones pueden modificar el comportamiento del vehículo en determinada maniobra. Con el modelo detallado de la red, mientras tanto, se puede hacer un análisis "*grano fino*" y determinar cómo una falla en el controlador CAN de alguno de los módulos conectados a la red afecta el funcionamiento de la misma. El análisis grano fino fue utilizado para identificar diferentes modos de malfuncionamiento de la comunicación sobre la red CAN. Estos resultados fueron utilizados después por el otro grupo de investigadores para hacer un análisis "grano grueso" y finalmente determinar la relación entre una falla en el controlador de acceso al bus CAN y la correspondiente degradación de la performance del vehículo durante una maniobra.

El modelo Simulink utilizado incluye no solamente módulos que modelan el comportamiento dinámico del vehículo en tiempo continuo, sino también módulos que simulan el comportamiento de la lógica de control del mismo, como son los bloques VDC (Vehicle Dynamic Controller) y ABS (Anti-lock Braking System). Estos módulos trabajan en tiempo discreto, intercambiando señales muestreadas y cuantificadas con sensores y actuadores. En el vehículo real esto se realiza intercambiando mensajes a través de la red CAN. En la simulación Matlab/Simulink este intercambio de mensajes está modelado como un bloque ad-hoc que implementa una cadena *Muestreo –> Cuantización –> Retardo – Reconstrucción de señal*. El modelo permite la inserción de fallas de diferente tipo inmediatamente después de la etapa de muestreo.

Ambos modelos incluyen los nodos correspondientes a las unidades de control (ECUs) conectadas a la red C-CAN del vehículo descrito en el apartado Parte II4.4. Los nodos involucrados son en total seis: Ángulo del volante (NAS), Freno (NFR), Freno asistido (NBA), Yaw (NYL), Caja de cambios automática (NCA), y Control de Motor (NCM).

Modelo FARM

Al diseñar un ambiente para inyección de fallas es común plantearlo en función de los conjuntos FARM (Faults, Activation, Readouts, Measures). Se describen a continuación estos cuatro conjuntos para la segunda serie de experimentos.

El conjunto F: Fallas

Para la inyección de fallas en el modelo detallado de la red se analizaron fallas del tipo bit-flip en un elemento de memoria del circuito. En este tipo de fallas se invierte el valor lógico almacenado en el registro en el período de reloj seleccionado como instante de inyección.

La falla queda determinada entonces por su ubicación (cuál es el flip-flop afectado) y por el instante de inyección.

El instante de inyección se eligió al azar con distribución uniforme a lo largo de toda la duración de la maniobra.

Para la ubicación de la falla también se utilizó distribución uniforme entre todos los elementos de memoria de uno de los nodos de la red considerado crítico. El nodo seleccionado es el Nodo Freno (NFR) que monitorea el estado de cada rueda del vehículo y genera la mayor parte del tráfico en la maniobra de viraje.

Conjunto A: tráfico destinado a Activar las fallas

Las fallas se inyectaron mientras el sistema emulaba la realización de una maniobra en la cual al vehículo en movimiento se le aplica un escalón en la entrada que suministra la posición del volante (maniobra “step-steer”), provocando un viraje cerrado. La maniobra está normalizada (ISO 7401:2003), tiene una duración total de 10 segundos y es conocida como particularmente crítica en términos de la seguridad del vehículo.

Los 6 nodos incluidos en el modelo acceden periódicamente al bus CAN para intercambiar valores actuales y deseados de variables del sistema. Existen 16 de esas variables que son relevantes en la maniobra analizada, que se codifican en 8 tipos de mensaje diferenciados por el ID de la trama CAN. En la tabla se indica para cada tipo de mensaje cuál es el nodo responsable de generarlo, cuáles nodos lo reciben, y cuál es el período de repetición. Todos los mensajes son tramas de 8 bytes excepto el mensaje Steering angle sensor que es de 6 bytes.

MESSAGE NAME	Period [ms]	NCM	NFR	NCA	NAS	NYL	NBA
ASR0	5		TX	RX	RX	RX	RX
Yaw Accelerator Sensor	10		RX			TX	
GEARMOT	10	RX		TX			RX
Steering Angle sensor	10		RX		TX		
ASR1	10	RX	TX	RX			RX
ASR2	10	RX	TX	RX			RX
MOT1	10	TX	RX	RX			RX
BA	10	RX	RX				TX

Fig. 11 Matriz de señales

Inicialmente se pensó en realizar una co-simulación, integrando ambos niveles de abstracción en un mismo experimento. De esa forma los mensajes a intercambiar en el modelo de descripción detallada de la red se construirían a partir de los valores que toman en los instantes de muestreo las variables relevantes en la simulación Matlab. Este objetivo resultó demasiado ambicioso para el tiempo de desarrollo disponible por lo que rápidamente se pasó a la arquitectura en dos fases descrita más arriba. La primera alternativa manejada para generar el tráfico a cursar por la red fue realizar una simulación en el modelo Matlab sin inyectar fallas y registrar los valores de todas las señales involucradas para construir con eso las tramas a intercambiar en el modelo detallado de la red. Sin embargo para identificar los diferentes modos de malfuncionamiento de la red CAN resultó igualmente efectivo intercambiar mensajes generados con la misma periodicidad, pero con valores de las variables generados automáticamente a partir del número de intervalo de muestreo de manera de simplificar la clasificación de las fallas.

Se utilizó la segunda versión del ambiente para inyección de fallas en el interior del controlador CAN, descripta en la segunda parte del apartado 6.1. Aprovechando la

periodicidad de los mensajes se dividió el experimento en intervalos iguales al menor período de repetición de los mensajes (5 ms) y se ubicaron los *checkpoints* al comienzo de cada intervalo, justo antes de comenzar la transmisión.

A efectos de emular la aplicación fue necesario agregar a *Castor* el código que describe el comportamiento de los nodos para transmisión y recepción. Esto implicó principalmente codificar las máquinas de estado para transmisión y recepción. En transmisión (ver Fig. 12), para cada intervalo de muestreo se implementó una lista de mensajes a transmitir por cada nodo. La transmisión del primer mensaje se solicita en la transición de esperoNuevoCiclo a esperoFinTx. Al terminar la transmisión del primer mensaje se pasa al estado envíoNuevoMsg donde se verifica si hay más mensajes para transmitir. Cuando se agota la lista de mensajes se retorna al estado inicial a esperar el comienzo del siguiente intervalo. La máquina de estados interactúa con el hardware de Pollux a través de una biblioteca muy sencilla que permite escribir el contenido del mensaje a transmitir, ordenar el inicio de la transmisión (señal tx_req) y consultar las señales de handshake (tx_done).

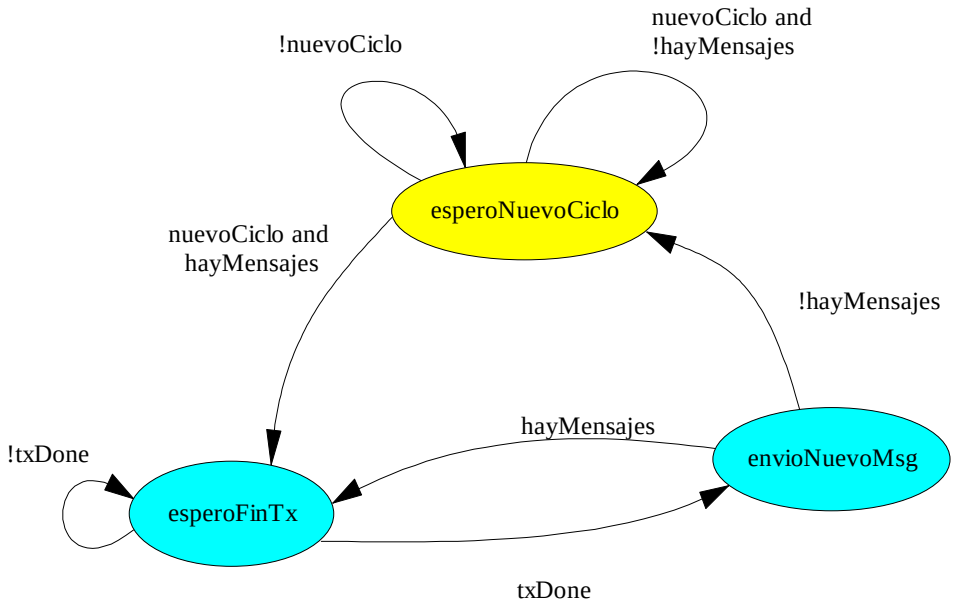

Fig. 12. Castor: Máquina de estados de Transmisión.

La máquina de estados para recepción (Fig. 13) es similar. La función del estado STATE1 es esperar que la señal de handshake rx_completed vuelva a estar inactiva. La máquina de estados debe encargarse de leer las tramas recibidas y, cuando la

matriz de señales indica que la trama debe ser leída por este nodo, verificar que el contenido y el instante de recepción son correctos como se describe más adelante.

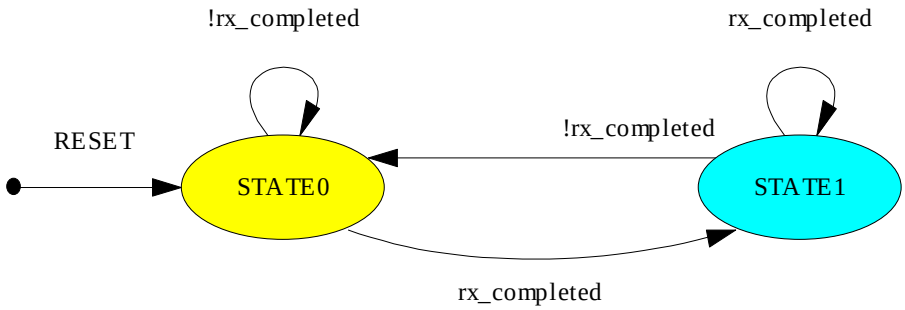

Fig. 13. Castor: Máquina de estados de Recepción

Se realizó una corrida de referencia o *golden run* sin inyectar fallas y se almacenó el estado de todos los Flip-Flops del sistema en cada checkpoint. Luego, para cada falla se comienza la ejecución a partir del checkpoint anterior al instante de inyección cargando el estado almacenado durante la *golden run.* Al llegar a cada checkpoint se observa el estado de los registros y se lo compara con el estado almacenado durante la *golden run*. El experimento continúa hasta que el estado coincide con el estado almacenado o se completa la duración de la maniobra completa. Nótese que en el caso de los experimentos en que la falla no produce ningún malfuncionamiento la duración del experimento es de sólo un intervalo de muestreo.

Clasificación de las fallas: Readouts y Medidas

Durante el experimento se registra cada vez que sucede alguno de los siguientes eventos:

- la solicitud a un nodo de transmitir una trama.
- la recepción de una trama por parte de un nodo.
- la inyección de la falla
- cada checkpoint
- la finalización del experimento

Para cada trama se comparó el valor recibido y el instante de recepción con los correspondientes valores e instantes registrados durante la corrida de referencia. Los efectos de la falla sobre la transmisión de cada trama se clasificaron en las siguientes clases, en orden creciente de gravedad:

- *E_OK*: la transmisión de la trama no se ve afectada por la falla
- *E_PERF*: luego de algunas tramas de error y retransmisiones la trama es finalmente correctamente recibida antes del comienzo del siguiente intervalo de muestreo. Sin embargo la recepción se completa en un instante posterior al esperado.
- *E_MISS*: al menos uno de los nodos que debe recibir la trama de acuerdo con la matriz de señales no la recibe antes del final del intervalo de muestreo.
- *E_WRONG*: al menos uno de los nodos que espera recibir la trama la recibe con un valor incorrecto en alguna de las variables relevantes para la maniobra.

Los malfuncionamientos del tipo E_PERF y E_WRONG se detectan y registran en el momento de recibir una trama, mientras que los del tipo E_MISS se analizan en el instante de checkpoint verificando si cada nodo ha recibido durante el último intervalo todas las tramas que debe recibir.

A efectos de la clasificación de la falla introducida en cada experimento, se tuvo en cuenta inicialmente el malfuncionamiento de mayor gravedad detectado a lo largo del mismo. Se registró además el instante y la clase del primer y último malfuncionamiento detectados para considerar la duración del efecto de la falla, aspecto fundamental a la hora de determinar las consecuencias de la falla sobre el sistema completo. En base a todo esto los efectos de las fallas se clasificaron en las siguientes categorías:

- *Sin efecto*. Todas las tramas intercambiadas durante la maniobra fueron transmitidas y recibidas correctamente.
- *Degradación de performance*: la falla provocó la recepción con retardo de alguna trama. Las tramas fueron recibidas con el contenido correcto pero arribaron más tarde que en la red sin fallas.

- *Trama faltante*: al menos uno de los nodos temporalmente no pudo recibir una o más tramas.

- *Datos incorrectos*: al menos uno de los nodos de la red recibió datos corruptos y esto no fue detectado por el protocolo.

- *Nodo bloqueado*: el nodo afectado por la falla detiene toda su actividad sobre el bus. El resto de los nodos siguen intercambiando información sin problemas sobre la red.

- *Bus bloqueado*: como resultado de la falla el bus se bloquea de forma que todos los nodos quedan impedidos de enviar o recibir tramas sobre el bus. Este caso que a priori podría existir no fue detectado en ninguno de los experimentos realizados.

Resultados

Se realizó una campaña de inyección de fallas compuesta por diez series de mil fallas cada una, totalizando 10.000 experimentos.

	#	%	% sobre el total de malfuncionamientos
Degradación de Performance	289	2,9	29,8
Trama faltante	59	0,6	6,1
Datos incorrectos	236	2,4	24.3
Nodo Bloqueado	387	3,9	39,9
Subtotal (fallas no silenciosas)	971	9,7	100,0
Sin efecto	9029	90,3	
Total	10000	100	

Fig. 14. Resultados de la inyección de fallas.

La tabla muestra los efectos observados que han sido clasificados en las categorías introducidas en el apartado anterior. Nótese que solamente el 9.7% de las fallas

inyectadas modifican el comportamiento de la red. Profundizando en el análisis de ese 9.7% se distinguieron los siguientes casos:

- 24.3% de los malfuncionamientos corresponden a la categoría *datos incorrectos* en que alguna trama es recibida sin que se detecte ninguna anomalía, pero la información contenida en la misma es errónea. En casi todos los casos esto se produce debido a fallas introducidas o bien en el buffer de salida donde residen los datos antes de que se arme la trama a transmitir, o bien en el nodo de recepción pero después que ha sido calculado y verificado el CRC de la trama. En ambos casos el error escapa a los mecanismos de detección provistos por el protocolo. Este tipo de fallas afecta solamente la transmisión de una trama. Debido a la naturaleza del sistema es poco probable que esto produzca un efecto importante sobre el comportamiento del vehículo: el valor erróneo de la señal será reemplazado por un valor correcto en el siguiente intervalo de muestreo (en 5 o 10 mseg. dependiendo de la variable afectada), y lo más probable es que la inercia del vehículo filtre el efecto de las acciones incorrectas que pudieran haberse tomado.

- 29.8% de los malfuncionamientos corresponden a la categoría *degradación de performance*. En estos casos la falla provoca una o varias retransmisiones de la trama, pero la misma es finalmente entregada a los nodos destino antes de terminar el intervalo de muestreo. La información correcta en este caso no se pierde, solamente llega a destino algunos microsegundos más tarde de lo esperado. En este caso la estrategia a ser aplicada al vehículo se decide en base a la información correcta por lo que el automóvil se comportará según lo esperado.

- 39.9% de los malfuncionamientos observados el nodo Freno detiene toda su actividad desde el instante de inyección de la falla hasta la finalización de la maniobra al menos (*Nodo bloqueado*). Este malfuncionamiento es particularmente crítico ya que provoca la interrupción del suministro de información actualizada sobre el estado de las ruedas, información que es esencial para el algoritmo que decide las acciones a ser tomadas para controlar el comportamiento del auto. Muy probablemente esto lleve a que el algoritmo

de control del vehículo tome acciones incorrectas que podrán llevar a una pérdida del control del vehículo por parte del conductor.

- 6.1% de los malfuncionamientos fueron clasificados como *Trama faltante*. Estos corresponden a una variedad de situaciones diferentes: en algunos casos se trató de errores similares a los clasificados como dato incorrecto pero afectando al ID de la trama, por lo que el mensaje fue descartado por el nodo receptor al no reconocerlo como el tipo de mensaje esperado. En otros casos correspondió a errores afectando a alguna máquina de estados interna del controlador que lo mantuvo bloqueado por un intervalo limitado, del orden de unos pocos intervalos de muestreo. Por los mismos motivos que en el caso de datos incorrectos, los efectos esperables sobre el comportamiento del vehículo son limitados.

- No se observaron situaciones de *Bus bloqueado*.

Efecto sobre el comportamiento del vehículo

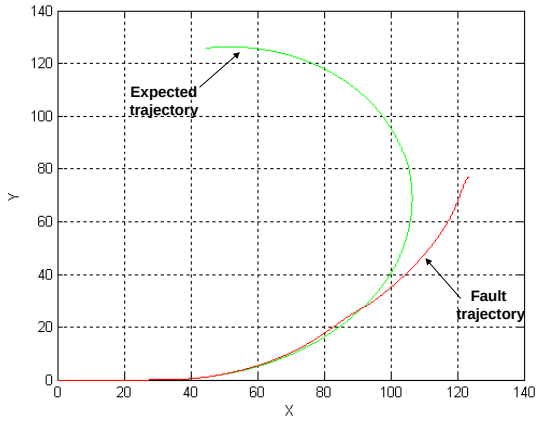

Fig. 15. Comparación entre la trayectoria esperada y la trayectoria con falla (reproducida de [18]).

Para analizar el efecto de los malfuncionamientos clasificados como *Nodo bloqueado* sobre el comportamiento del vehículo, fueron realizadas simulaciones por Mattia Ramasso y Simonluca Tosato utilizando el modelo dinámico del vehículo con Matlab/Simulink. Se encontró que en el intervalo comprendido entre 1.00 y 1.25 segundos desde el inicio de la maniobra el comportamiento del vehículo es especialmente sensible a la situación de nodo bloqueado del nodo Freno. Para ilustrar la criticidad de estas fallas se incluye un gráfico calculado con el modelo dinámico del vehículo. En Fig. 15 se muestran la trayectoria esperada y la trayectoria resultante de la inyección de la falla. En los ejes se representa la posición del vehículo en metros.

Recursos utilizados y performance obtenida

La campaña de inyección de fallas sobre el modelo detallado de la red CAN descrita en los apartados anteriores tuvo una duración aproximada de 6 horas y media. La plataforma estaba formada por un computador personal que aloja una placa con FPGA insertada en un slot PCI. La placa programable utilizada fue una Alphadata ADM-XRC, equipada con un FPGA Virtex 2000e de Xilinx.

El modelo completo incluyó unas 2,000 líneas de código C para implementar Castor y unas 4,000 líneas de código VHDL para Pollux. Utilizando las herramientas de Xilinx Pollux se sintetizó para la Virtex 2000E de la placa.

El porcentaje de utilización del chip para la red de 6 nodos implementada fue bajo (8% de las look-up tables y 4% de los flip-flops) de donde se deduce que la misma plataforma puede ser utilizada para diseños con un número bastante mayor de nodos.

Conclusiones de la segunda serie de experimentos

La segunda serie de experimentos realizada durante este trabajo permitió detectar en forma eficiente el subconjunto de fallas de mayor criticidad. Estas fallas pueden modificar fuertemente el comportamiento del vehículo afectando por consiguiente la seguridad tanto de los ocupantes del vehículo como de terceros. Por ese motivo es importante identificar estas fallas lo antes posible en la fase de diseño del sistema a efectos de realizar las correcciones necesarias. El ambiente presentado en este capítulo desarrollado en cooperación con otros investigadores es una herramienta idónea para ese fin ya que permite evaluar el efecto de las fallas sobre el funcionamiento del sistema observado a un alto nivel de abstracción.

Capítulo 8

Conclusiones y trabajos futuros

Cuando se realiza un experimento de inyección de fallas, en la mayoría de los casos no es suficiente con verificar que existen diferencias entre las salidas del sistema con y sin falla inyectada para poder clasificar sus efectos. Dicho de otra forma, diferentes secuencias en las salidas del sistema pueden mantenerse dentro de las especificaciones y ser por tanto igualmente aceptables. Esto hace que sea muy difícil automatizar totalmente experimentos de inyección de fallas ya que para clasificar el efecto de las mismas hace falta tener cierto conocimiento del sistema particular que se está ensayando.

Otra consecuencia de lo señalado en el párrafo anterior es que para saber si el comportamiento del sistema frente a una falla es o no un malfuncionamiento, es necesario realizar el análisis en términos del funcionamiento del mismo en alto nivel, mientras que a menudo interesa analizar el efecto de las fallas modeladas en bajo nivel. Si se quiere realizar la inyección de fallas durante la fase de desarrollo del sistema, cuando todavía no se cuenta con un ejemplar funcionante del sistema, nos encontramos frente a un problema: si analizamos todo el sistema con el grado de detalle necesario para modelar la falla, la ejecución del experimento se torna computacionalmente irrealizable. Resulta crítico entonces circunscribir el análisis detallado a la parte del sistema en la cual se inyecta la falla, y utilizar modelos de más alto nivel en el resto del sistema. El uso de prototipado hardware sobre FPGAs para acelerar la ejecución en la parte del modelo modelada en detalle, y las herramientas desarrolladas en el presente trabajo para interconectar los modelos de diferente nivel resultan imprescindibles para poder mantener el tiempo de ejecución de los experimentos dentro de límites razonables.

En ese sentido se modeló la red CAN en nivel RTL y a cada nodo de la red como un autómata descrito o bien lenguaje en C o bien en VHDL comportamental. Se utilizó el prototipado sobre FPGAs para acelerar la ejecución del modelo de la red en nivel RTL. Se desarrollaron herramientas para poder cambiar modularmente la funcionalidad en los nodos, tanto para la versión VHDL como para la versión en

lenguaje C, y comunicarlos en forma estrecha con el modelo RTL de la red. Se utilizó dicho ambiente para identificar los modos de malfuncionamiento producidos a nivel de la comunicación entre los nodos para una aplicación determinada y, trabajando en cooperación con otro grupo dentro del Politecnico di Torino, se analizaron sus efectos sobre un modelo del comportamiento dinámico del vehículo, verificándose que pueden tener consecuencias muy graves sobre la seguridad del vehículo.

8.1. Resultados obtenidos.

Se listan a continuación los principales resultados y productos obtenidos a lo largo del trabajo en la presente tesis.

- Un primer producto tangible es el del core-IP para el controlador de acceso al bus CAN. Dicho controlador fue desarrollado siguiendo la especificación de CAN 2.0b, y fue escrito en VHDL, fue sintetizado sobre chips Virtex de Xilinx y probado en los experimentos de inyección de fallas. En el diseño se evitó utilizar bloques de las bibliotecas del fabricante para obtener un código lo más portable posible.

- Se desarrolló un bloque *saboteur* y las interfaces necesarias para accederlo desde fuera del circuito. El bloque *saboteur* una vez adosado a un registro del circuito permite cargarle el valor que se desee o complementarlo. La incorporación de los *saboteurs* y la lógica de interfaz adicional a la descripción VHDL del controlador CAN fue realizada en forma manual. Este proceso se ve facilitado si en el código VHDL original las entradas a los registros están evaluadas explícitamente, en cuyo caso se estima que sería relativamente sencillo automatizarlo.

- Se desarrolló el conjunto de herramientas descrito [6.3] para la realización de campañas de inyección de fallas por emulación en circuito. Dichas herramientas permiten montar experimentos de inyección de fallas para diferentes sistemas compuestos por varios nodos comunicados a través de un bus CAN. Puede extenderse a otros protocolos de red si se cuenta con la descripción RTL del controlador correspondiente y se le incorporan los *saboteurs* descriptos en el punto anterior.

- Se identificaron modos de malfuncionamiento asociados a fallas en los registros internos del controlador de acceso al bus CAN.

- Se realizaron varias campañas de inyección de fallas para analizar el efecto de fallas del tipo bit-flip en la totalidad de los registros internos del controlador de acceso al bus CAN con una performance muy buena en términos de la velocidad de ejecución. Se identificaron los modos de malfuncionamiento producidos a nivel de la comunicación entre los nodos.

8.2. Trabajos futuros.

Se listan a continuación varios temas y líneas de trabajo a futuro. Algunos son simplemente aspectos a terminar para un mejor aprovechamiento del trabajo realizado. Otros en cambio son nuevos temas en los cuales parece interesante seguir avanzando.

- Controlador CAN. Aquí en primer lugar hace falta completar el desarrollo incorporando los mecanismos de confinamiento de fallas. También interesa agregarle funcionalidades a la interfaz hacia la capa de aplicación, habituales en los controladores disponibles comercialmente: FIFOs que permitan encolar la recepción o envío de varias tramas, filtrado de tramas según matcheo con patrones en el campo identificador, etc.

- Hay espacio para mejorar bastante el proceso de incorporación de los bloques *saboteurs* y la lógica de inyección de fallas a un circuito. Un primer paso simple sería formalizar reglas de estilo para el código VHDL en que debe describirse un circuito para simplificar el proceso. El resultado final deseado es la automatización del mismo.

- Las herramientas software para la inyección de fallas y modelado del comportamiento de los nodos de la red fueron escritas en lenguaje C, y teniendo siempre presente la velocidad de ejecución. La utilización de metodologías y lenguajes de programación orientada a objetos tendría sin dudas un efecto muy beneficioso en la mantenibilidad y facilidad de uso de estas herramientas, disminuyendo fuertemente el tiempo necesario para la preparación de cada nueva campaña de inyección de fallas. Tendría

seguramente también un efecto negativo sobre la velocidad de ejecución de cada experimento. Parece interesante entonces en una primera etapa estimar cuantitativamente estas ventajas y desventajas, y si se evalúa conveniente migrar las herramientas a una biblioteca de clases p. ej. en C++ para simplificar su uso.

- Integración de modelos en Simulink. No fue posible en el alcance de esta tesis integrar en un experimento conjunto la simulación del comportamiento dinámico de un vehículo modelada en Matlab/Simulink con la emulación de la red CAN realizada sobre un prototipo hardware sintetizado en un FPGA.

- Como fue comentado en Parte II, FlexRay se perfila como el sucesor de CAN en aplicaciones automotrices con requerimientos fuertes de *dependabilidad*. Tiene interés entonces extender el análisis presentado en esta tesis a redes FlexRay, y comparar resultados con los obtenidos para redes CAN.

8.3. Comentarios finales

Los resultados numéricos de los experimentos de inyección de fallas presentados en este trabajo están fuertemente influidos por lo bien o mal que se comporta frente a fallas el controlador CAN utilizado. Dicho controlador no fue diseñado con el objetivo de hacerlo tolerante a fallas y no hubo un interés especial por hacer una evaluación cuantitativa de sus atributos de *dependabilidad*.

Entonces, los resultados más fuertes del presente trabajo no son los resultados numéricos, que no son generalizables, sino por un lado los resultados cualitativos en cuanto a los efectos que puede tener una falla en el controlador CAN sobre el comportamiento del vehículo, y por otro lado la metodología de modelado en diferentes niveles y las herramientas desarrolladas tendientes a integrar esos diferentes niveles en un mismo experimento.

REFERENCIAS

[1] Cheng, K.; Huang, S. & Dai, W.; Fault Emulation: A New Methodology for Fault Grading; IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, *1999, vol. 18,* **pp.** 1487-1495

[2] Hsueh, M.; Tsai, T. & Iyer, R.K.; Fault Injection Techniques and Tools; *IEEE Computer,* **1997,** *vol. 30,* no. *4*, pp. 75-82

[3] Wieler, R.; Zhang, Z. & McLeod, R.; Emulating static faults using a Xilinx based emulator; IEEE Symposium on FPGA's for Custom Computing Machines (FCCM '95); *IEEE Computer Society;* **1995**

[4] Navet, N.; Song, Y.; Simonot-Lion, F. & C., W.; Trends in Automotive *Communication Systems;* Proceedings **of** the IEEE, 2005, vol. 96, no. *6*, pp. 1204-1223

[5] Avizienis, A.; Laprie, J.; *Randell, B. &* Landwehr, C.; Basic Concepts and Taxonomy of Dependable and Secure Computing; *Dependable and Secure Computing, IEEE Transactions on,* **2004,** *vol.* 1, no. *1,* pp. 11-33

[6] P. Civera, P.; Macchiarulo, L.; Rebaudengo, M.; Sonza Reorda, M. & Violante, M.; An FPGA-based *approach for speeding-up Fault Injection campaigns on safety-critical* circuits; *Journal of Electronic Testing: Theory and Applications, Kluwer Academic Publishers;* **2002,** *vol.* 18, no. *3,* pp. 261-271

[7] *Navet, N.; Song, Y.* & Simonot, F.; Worst-Case Deadline Failure Probability in Real-Time Applications Distributed over CAN (Controller *Area Network);*

Journal of Systems Architecture, Elsevier **Science,** *2000, vol.* 46, no. *7,* pp. 607-617

[8] Kanawati, G.A.; Kanawati, N.A. & Abraham, J.A.; FERRARI: A Flexible Software-Based Fault and Error System; IEEE Transactions on computers, **1995,** vol. 44, no. *2,* pp. 248-260

[9] Arlat, J.; *Aguera, M.; Amat, L.; Crouzet, Y.; Fabre, J.; Laprie, J.; Martins, E.* & Powell, D.; Fault Injection for Dependability Validation: A Methodology and Some Applications; *IEEE Transactions on Software Engineering,* **1990,** *vol.* 16, no. *2,* pp. 166-182

[10] Laprie, J.; Dependability: Basic concepts and terminology in English, French, German, Italian and Japanese, Dependable Computing and Fault Tolerance; Springer-Verlag; 1992

[11] Arlat, J.; Boué, J.; Crouzet, Y.; *Jenn, E.; Aidemark,* J.; Folkesson, P.; Karlsson, J.; Ohlsson, J. & Rimén, M.; Benso, A. & Prinetto, P. *(ed.);* Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation; 4.2: MEFISTO: A Series of Prototype Tools for Fault Injection into VHDL Models; *Kluwer;* **2003,** pp. 177-193

[12] *Arlat, J.;* Fabre, *J.;* Rodríguez, M. & Salles, F.; Benso, A. & Prinetto, P. (ed.); *Fault Injection Techniques and* **Tools** *for Embedded* Systems Reliability Evaluation; 3.3 MAFALDA: A Series of Prototype Tools for the Assessment of Real Time COTS Microkernel-based Systems; *Kluwer;* **2003,** pp. 141-156

[13] Baldini, A.; Benso, A. **&** Prinetto, P.; Benso, A. & Prinetto, P. (ed.); Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation; 3.1 "BOND": An Agents-based *Fault Injector for Windows NT;* **Kluwer;** *2003, pp.* 111-123

[14] Costa, D.; Madeira, H.; Carreira, J. & Silva, J.G.; Benso, A. & Prinetto, P. *(ed.);* Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation; 3.2 XCEPTION: A Software Implemented Fault Injection Tool; *Kluwer;* **2003,** pp. 124-139

[15] Gil, D.; Baraza, J.C.; *Garcia,* J. & Gil, P.J.; Benso, A. & Prinetto, P. (ed.); Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation; 4.1 VHDL Simulation-based Fault Injection Techniques; *Kluwer;* **2003,** pp. 159-176

[16] Gil, *P.;* Blanc, S. & Serrano, J.J.; Benso, A. & Prinetto, P. (ed.); Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation; 2.1 *Pin-Level* Hardware Fault Injection Techniques; Kluwer; 2003, pp. 63-79

[17] Yu, Y. & Johnson, B.W.; Benso, A. & Prinetto, P. (ed.); Fault Injection Techniques and Tools *for* Embedded Systems Reliability Evaluation; 1.1 Fault Injection Techniques; *Kluwer;* **2003,** pp. 7-39

[18] Corno, F.; Pérez Acle, J.; Sonza Reorda, M. & Violante, M.; Validation of the dependability of CAN-based networked systems; **IEEE** High-level Design Validation and Test Workshop; 2004, pp. 161-164

[19] Corno, F.; Pérez Acle, J.; Sonza Reorda, M. & Violante, M.; A multi-level *approach* to the dependability analysis of CAN networks for automotive applications; International Conference Integrated Chassis Control(ICC); **2004**

[20] Corno, F.; Pérez Acle, J.; Sonza Reorda, M. & Violante, M.; A Multi-Level Approach to *the Dependability Analysis of* Networked **Systems** *Based* on the CAN Protocol; SBCCI '04: Proceedings of the 17th *symposium* **on** Integrated circuits and system design; *ACM Press;* **2004,** pp. 71-75

[21] Pérez Acle, J.; Sonza Reorda, M. & Violante, M.; Dependability Analysis of CAN Networks: an emulation-based approach; 18th IEEE International Symposium on Defect **and** Fault Tolerance in VLSI Systems (DFT'03); IEEE Computer Society *Press;* **2003,** pp. 537-544

[22] Pérez Acle, J.; Sonza Reorda, M. & Violante, M.; Accurate Dependability Analysis of CAN-based Networked Systems; Proceedings **of** 16th Symposium on Integrated Circuits and Systems Design (SBCCI); IEEE Computer Society Press; *2003,* **pp.** 337-342

[23] Velardocchia, M. & Sorniotti, A.; Vehicle Dynamics Control (VDC) and Active Roll Control (ARC) Integration to Improve Handling and Comfort; Proc. of International Conference on Vehicle and Systems Progress, Volgograd (Russia); **2002**

[24] Avizienis, A.; Laprie, J. & Randell, B.; Fundamental Concepts of Dependability; Third Information Survivability Workshop -- ISW-2000; IEEE; 2000

[25] Rufino, J.; Verissimo, P.; Arroz, G.; Almeida, C. & Rodrigues, *L.;* Fault-Tolerant Broadcasts in CAN; Symposium on Fault-Tolerant Computing; **1998,** pp. 150-159

[26] Rufino, J.; An Overview of the Controller Area Network; Proceedings of the CiA Forum CAN for Newcomers; 1997

[27] Verissimo, P.; Rufino, J. & Ming, L.; *How* hard is hard real-time communication on field-buses?; Digest of Papers, The 27th International Symposium on Fault-Tolerant Computing Systems; **1997,** pp. 112-121

[28] Rufino, J. & Veríssimo, P.; A STUDY ON THE INACCESSIBILITY CHARACTERISTICS OF THE CONTROLLER AREA NETWORK; International *CAN* **Conference,** 2nd iCC, Proceedings 1995; 1995

[29] Tindell, K.; Burns, A. & Wellings, A.; Calculating Controller Area Network (CAN) Message Response Times; Proceedings of IFAC DCCS`94; **1994,** pp. 35-40

[30] Laprie, J.; Dependable computing and fault tolerance: concepts and terminology; Proc. 15th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-15); 1985, *pp. 2-11*

[31] *FlexRay_Consortium; FlexRay Communication System, Protocol* Specification, *Version* 2.0; 2004

[32] OSEK_Consortium; OSEK/VDX Communication, Version 3.0.3; 2004

[33] ISO 7401:2003 - Road vehicles -- Lateral transient response test methods -- Open-loop test methods; 2003

[34] Time-Triggered Protocol TTP/C, High-Level Specification Document, Protocol Version 1.1; 2003

[35] Kopetz, H. & al.; Specification of *the TTP/A* Protocol; 2002

[36] International Standard Organization, 11898-4, Road Vehicles - Controller Area Network (CAN) - Part 4: Time-Triggered *Communication;* **2000**

[37] CAN Specification. Version 2.0; 1991

[38] Mohor, I. & Shaheen, S.; Opencores Can *protocol controller project. http://www.opencores.org/projects/can/;* **2003**

[39] Juárez, J. & Bustamante, P.; Interfase Bus Can. Proyecto final del curso Diseño Lógico 2; **2002**

[40] Doll, S.; VHDL Verification course at http://www.stefanvhdl.com/;

[41] Alphadata home page http://www.alphadata.co.uk;

[42] Bosch's Controller Area Network Homepage at http://www.can.bosch.com/;

[43] CAN in Automation *(CiA) Homepage at http://www.can-cia.org/;*

[44] CANKingdom International Homepage at http://www.cankingdom.org/;

[45] *LIN* Local Interconect Network Homepage at http://www.lin-subbus.org/;

[46] Model Technology Homepage at http://www.model.com/;

[47] Open DeviceNet Vendor Association (ODVA) Homepage at http://www.odva.org/;

[48] OSEK Consortium homepage http://www.osek-vdx.org/;

[49] TTTech - Time-Triggered Technology home page at http://www.tttech.com/;

[50] Web server of the **Real-Time** Systems Research Group at the Vienna University of Technology. http://www.vmars.tuwien.ac.at/;

[51] Rufino, J.; Computational System for Real-Time Distributed Control. Phd Thesis.; Technical University of Lisbon - **Instituto** Superior *Técnico,* **2002**

[52] Gaujal, B. & Navet, N.; RR-4603 - Fault Confinement **Mechanisms** of the CAN Protocol : Analysis and Improvements; Rapport **de** recherche; Rapport *de recherche de l'INRIA - Lorraine , Equipe : TRIO,* **2002,** no. *RR-4603*

[53] Avizienis, A.; Laprie, J. & Randell, B.; Fundamental Concepts of Dependability; LAAS, 2001, no. LAAS Report no. *01-145*

[54] Avizienis, A.; Laprie, J.; Randell, B. & Landwehr, C.; Garantía de funcionamiento: Conceptos básicos y terminología. traducción por Pedro GIL VICENTE de "Dependability: Basic concepts and terminology"; *DISCA-UPV. informe interno.,* **1996**

ANEXOS

Anexo 1

Dependabilidad. Índice de términos y su correspondencia con los términos en idioma inglés.

En 1992 los trabajos de Avizienis, Laprie y Randell que sistematizan los conceptos y la terminología relacionados con la *dependabilidad* de los sistemas de cómputo fue traducida a varios idiomas (*Dependability: Basic concepts and terminology in English, French, German, Italian and Japanese* [10]). En español sin embargo el panorama es todavía algo confuso. Al comenzar a escribir esta tesis se procuró encontrar alguna traducción de la terminología al español, incluso a través de una consulta al Dr. Laprie, pero sin resultados positivos. Por ese motivo se realizó la traducción de parte de los trabajos de Laprie presentada en el Capítulo 1 y la lista de definiciones y correspondencia con los términos en los otros idiomas que se presentan a continuación. Se tomó a veces como ayuda las versiones en francés e italiano.

Cerca del final del trabajo en esta tesis se estableció contacto con Pedro Gil de la Universidad Politécnica de Valencia. Pedro Gil había realizado un esfuerzo anterior por llevar la terminología a idioma español, incluyendo una traducción completa de "*Basic concepts and terminology*", y se encuentra en este momento trabajando en una nueva versión de la misma. Los términos utilizados en esta tesis difieren de los utilizados por Gil.

Para designar "*dependability*" se optó por introducir el vocablo *dependabilidad* a pesar de no existir en español. La otra opción manejada fue "*certeza de funcionamiento*", similar a "*sureté de fonctionnement*" utilizado en francés pero evitando la confusión de los varios significados del vocablo *seguridad*. Se reservó *confiabilidad* para designar al atributo *reliability*.

Para designar *failure* se utilizó el término *malfuncionamiento*, que refleja bien el significado de "desviación con respecto al cumplimiento de la especificación", pero no del todo bien el de "transición de entrega de servicio correcto a entrega de servicio incorrecto". El término utilizado por Gil, *avería*, refleja en mi opinión más la causa de

un malfuncionamiento que el malfuncionamiento mismo ("avería: daño que impide el funcionamiento de un aparato, instalación, vehículo, etc." según la Real Academia Española)

Otros términos problemáticos son *safety* y *security*. Personalmente no había reparado en la variedad de significados que tiene en idioma español el término *seguridad*. El problema fue resuelto en la terminología en francés agregando un calificativo separado por guión (*sécurité-innocuité*, *sécurité-confidencialité*). En este trabajo se optó por utilizar como calificativo el término original en inglés en lugar de *confidencialidad* ya que el significado de *security* es más amplio que el significado de confidencialidad, pero claramente no debería ser una solución definitiva.

A continuación se listan las definiciones de los términos más importantes, traducidas más o menos textualmente del texto de Laprie. Se indica entre paréntesis el término original en inglés. Para algunos términos se presentan acepciones adicionales de uso común. Al final se listan en forma de tabla los términos en inglés ordenados alfabéticamente, junto con los términos correspondientes en español, francés e italiano.

Amenazas a la dependabilidad (threats, impairments)

Circunstancias no deseadas, pero no inesperadas, que provocan o pueden dar como resultado ausencia de *dependabilidad*. Fallas, errores y malfuncionamientos.

Atributos de la dependabilidad (attributes)

Atributos que permiten valorar la calidad de un sistema que resulta de la oposición entre las amenazas y los medios. Confiabilidad, disponibilidad, mantenibilidad, seguridad-safety, seguridad-security.

Cobertura (coverage)

Medida de la representatividad de las situaciones a las que se somete un sistema durante su validación, en comparación con las situaciones a las que se verá enfrentado durante su vida operativa.

Otra definición: la probabilidad condicional de que el sistema se recupere, dado que se ha producido una falla.

Confiabilidad (reliability)

Dependabilidad con respecto a la continuidad del servicio. Medida de la entrega continua de servicio correcto. Medida del tiempo hasta una falla.

Otra definición [17]: la probabilidad condicional de que el sistema se comporte correctamente en todo el intervalo [*to*, *t*], dado que el sistema funcionaba correctamente en el instante *t* = *to*.

Dependabilidad (dependability)

La capacidad de un sistema de proveer un servicio en el cual se puede confiar en forma justificada.

Disponibilidad (availability)

Dependabilidad con respecto a estar listo para ser utilizado. Medida de la entrega de servicio correcto con respecto a la alternancia entre servicio correcto y servicio incorrecto.

Otra definición [17]: la probabilidad de que un sistema esté operando correctamente y esté disponible para realizar su función en el instante de tiempo *t*.

Error (error)

Un estado del sistema que puede provocar un subsiguiente malfuncionamiento.

Error detectado (detected error)

Error reconocido como tal por un algoritmo o mecanismo de detección.

Error latente (latent error)

Error no reconocido como tal.

Falla (fault)

Causa hipotética o asignada de un error.

Falla activa (active fault)

Falla que está produciendo un error.

Falla dormida (dormant fault)

Falla interna no activada aún

Malfuncionamiento (failure)

Desviación del servicio entregado respecto al cumplimiento de las especificaciones. Transición de entrega de servicio correcto a entrega de servicio incorrecto.

Medios para alcanzar la dependabilidad (means)

Métodos y técnicas que permiten: a) dar a un sistema la capacidad de entregar un sistema en el cual se puede depositar confianza, y b) ganar confianza en esa capacidad del sistema.

Modo de evitar fallas (fault avoidance)

Métodos y técnicas que procuran producir un sistema libre de fallas. Prevención y remoción de fallas.

Predicción de fallas (forecasting)

Métodos y técnicas con el fin de estimar la cantidad actual, la incidencia futura y las consecuencias de fallas.

Prevención de fallas (prevention)

Métodos y técnicas destinados a evitar la ocurrencia o introducción de fallas.

Remoción de fallas (removal)

Métodos y técnicas destinados a reducir la presencia (en número y seriedad) de fallas.

Restauración de servicio (service restoration))

Transición de entrega de servicio incorrecto a entrega de servicio correcto.

Seguridad-safety (safety)

Dependabilidad con respecto a la no ocurrencia de fallas catastróficas. Medida de la entrega continua o bien de servicio correcto, o bien de servicio incorrecto luego de una falla benigna. Medida del tiempo hasta una falla catastrófica.

Seguridad-security (security)

Dependabilidad con respecto a prevenir manejo de información o acceso no autorizados.

Servicio (service)

Comportamiento del sistema, tal como lo percibe el usuario del mismo.

Test (testing)

Verificación dinámica realizada aplicando valores en las entradas.

Tolerancia a fallas (fault tolerance)

Métodos y técnicas con el fin de proveer un sistema que cumpla con sus especificaciones a pesar de la presencia de fallas.

Tratamiento de fallas (fault treatment)

Acciones tomadas a efectos de evitar que una falla sea reactivada.

Verificación (verification)

El proceso de determinar si un sistema adhiere a ciertas propiedades que pueden ser a) generales, independientes de la especificación, o b) específicas, deducidas a partir de la especificación.

Verificación estática (static verification)

Verificación realizada sin ejercitar al sistema.

inglés	**español (utilizados en esta tesis)**	**español [54]**	**francés**	**italiano**
active fault	falla activa	fallo activo	faute active	guasto attivo
attributes of dependability	atributos de la dependabilidad	atributos de la garantía de …	attributs de la s^ureté de …	attributi della garanzia di …
availability	disponibilidad	disponibilidad	disponibilité	disponibilita'
avoidance (fault ~)	modo de evitar fallas	evitación de fallos	evitement des fautes	modo di evitare il guasto
coverage	cobertura	cobertura	couverture	copertura
dependability	dependabilidad	garantía de funcionamiento	s^ureté de fonctionnement	garanzia di funzionamento
detected error	error detectado	error detectado	erreur détectée	errore rilevato
dormant fault	falla dormida	fallo dormido	faute dormante	guasto inattivo
error	error	error	erreur	errore
failure	malfuncionamiento	avería	défaillance	fallimento, malfunzionamento
fault	falla	fallo	faute	guasto
forecasting (fault ~)	predicción de fallas	predicción de fallos	prévision des fautes	previsione del guasto
impairments to dependability	amenazas	impedimentos	entraves	impedimenti
latent error	error latente	error latente	erreur latente	errore latente
masking (fault ~)	enmascaramiento de fallas	enmascaramiento de fallos	masquage de faute	mascheramento del guasto
means for dependability	medios para alcanzar la …	medios para la garantia de …	moyens	mezzi
prevention (fault ~)	prevención de fallas	prevención de fallos	prévention des fautes	prevenzione del guasto
reliability	confiabilidad	fiabilidad	fiabilité	affidabilita'
removal (fault ~)	remoción de fallas	eliminación de fallos	elimination des fautes	eliminazione del guasto, rimozione
restoration (service ~)	restauración de servicio	restauración del servicio	restauration du service	ripristino del servizio
safety	seguridad-safety	seguridad-inocuidad	sécurité-innocuité	sicurezza di funzionamiento
security	seguridad-security	seguridad-confidencialidad	sécurité-confidencialité	sicurezza-confidenzialita'
service	servicio	servicio	service	servizio
static verification	verificación estática	verificación estática	vérification statique	verifica statica
testing	test	test	test	test, collaudo
tolerance (fault ~)	tolerancia a fallas	tolerancia a fallos	tolérance aux fautes	tolleranza al guasto
treatment (fault ~)	tratamiento de fallas	tratamiento de fallas	traitement de faute	trattamento del guasto
verification	verificación	verificación	vérification	verifica

Anexo 2

Trabajos publicados

8.1. 16th Symposium on Integrated Circuits and Systems Design (SBCCI) 2003

Página de la conferencia: http://www.inf.ufrgs.br/chipinsampa/

Proceedings:
http://ieeexplore.ieee.org/xpl/tocresult.jsp?isnumber=27627&isYear=2003

Pérez Acle, J.; Sonza Reorda, M. & Violante, M.; *Accurate Dependability Analysis of CAN-based Networked Systems*; Proceedings of 16th Symposium on Integrated Circuits and Systems Design (SBCCI); *IEEE Computer Society Press;* **2003,** 337-342

Abstract: Computer-based systems where several nodes exchange information via suitable network interconnections are today exploited in many safety-critical applications, like those belonging to the automotive field. Accurate dependability analysis of such a kind of systems is thus a major concern for designers. In this paper we present an environment we developed in order to assess the effects of faults in CAN-based networks. We developed an IP core implementing the CAN protocol controller, and we exploited it to set-up a network composed of several nodes. Thanks to the approach we adopted we were able to assess via simulation-based fault injection the effects of faults both in the bus used to carry information and inside each CAN controller, as well. In this paper, we report a detailed description of the environment we set-up and we present some preliminary results we gathered to assess the soundness of the proposed approach.

8.2. 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03) 2003

Página de la conferencia: http://tima.imag.fr/Conferences/dft-2003/

Proceedings:
http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/proceedings/dft/&toc=comp/proceedings/dft/2003/2042/00/2042toc.xml

Pérez Acle, J.; Sonza Reorda, M. & Violante, M.; *Dependability Analysis of CAN Networks: an emulation-based approach*; 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03); *IEEE Computer Society Press;* **2003,** 537-544

Abstract: Today many safety-critical applications are based on distributed systems where several computing nodes exchange information via suitable network interconnections. An example of this class of applications is the automotive field, where developers are exploiting the CAN protocol for implementing the communication backbone. The capability of accurately evaluating the dependability properties of such a kind of systems is today a major concern. In this paper we present a new environment that can be fruitfully exploited to assess the effects of faults in CAN-based networks. The entire network is emulated via an ad-hoc hardware/software system that allows easily evaluating the effects of faults in all the network components, namely the network nodes, the protocol controllers and the transmission channel. In this paper, we report a detailed description of the environment we set-up and we present some preliminary results we gathered to assess the soundness of the proposed approach.

8.3. 17th Symposium on Integrated Circuits and Systems Design (SBCCI) 2004

Página de la conferencia: http://www.cin.ufpe.br/~chiponthereefs/

Proceedings:
http://ieeexplore.ieee.org/xpl/tocresult.jsp?isnumber=29841&isYear=2004

Corno, F.; Pérez Acle, J.; Sonza Reorda, M. & Violante, M.; *A Multi-Level Approach to the Dependability Analysis of Networked Systems Based on the CAN Protocol*; SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design; *ACM Press;* **2004,** 71-75

Abstract: Safety-critical applications are now common where both digital and mechanical components are deployed, as in the automotive fields. The analysis of the dependability of such systems is a particularly complex task that mandates modeling capabilities in both the discrete and in the continuous domains. To tackle this problem a multi-level approach is presented here, which is based on abstract functional models to capture the behavior of the whole system, and on detailed structural models to cope with the details of system components. In this paper we describe how the interaction between the two levels of abstraction is managed to provide accurate analysis of the dependability of the whole system. In particular, the proposed technique is shown to be able to identify faults affecting the CAN network whose effects are most likely to be critical for vehicle?s dynamic. Exploiting the information about the effects of these faults, they can then be further analyzed at the higher level of details.

8.4. IEEE High-level Design Validation and Test Workshop (HLDVT) 2004

Página de la conferencia: http://www.hldvt.com/04/

Proceedings:
http://ieeexplore.ieee.org/xpl/tocresult.jsp?isnumber=30870&isYear=2004

Corno, F.; Pérez Acle, J.; Sonza Reorda, M. & Violante, M.; *Validation of the dependability of CAN-based networked systems*; IEEE High-level Design Validation and Test Workshop; **2004,** 161-164

Abstract: The validation of CAN-based systems is mandatory to guarantee the dependability levels that international standards impose in many safety-critical applications. In this extended abstract we present an environment to study how soft errors affecting the memory elements of network's nodes may alter the dynamic behavior of a car executing a maneuver. The experimental evidence of the effectiveness of the approach is reported on a case study.

8.5. International Conference Integrated Chassis Control (ICC) 2004

Corno, F.; Pérez Acle, J.; Sonza Reorda, M. & Violante, M.; *A multi-level approach to the dependability analysis of CAN networks for automotive applications*; International Conference Integrated Chassis Control(ICC); Nov. **2004**; pp. 10-12.

Abstract: The validation of networked systems is mandatory to guarantee the dependability levels that international standards impose in many safety-critical applications. In this paper we present an environment to study how soft errors affecting the memory elements of network nodes in CAN-based systems may alter the dynamic behavior of a car. The experimental evidence of the effectiveness of the approach is reported on a case study.

Texto completo: http://www.cad.polito.it/FullDB/exact/icc04.html