UNIVERSIDAD POLITÉCNICA DE VALENCIA

DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

VALIDACIÓN POR INYECCIÓN DE FALLOS EN VHDL DE LA ARQUITECTURA TTA

Tesis Doctoral

Presentada por

D. Joaquín Gracia Morán

Dirigida por

Dr. D. Daniel Antonio Gil Tomás

Dr. D. Pedro Joaquín Gil Vicente

Valencia, 2004

Y Marga estuvo allí

“Aventuras, emoción, no es lo que busca un caballero Jedi”
Bob el Silencioso, Mallrats, 1995

“We want, information, information, information
Who are you?
The new number 2
Who is number 1
You are number 6
I am not a number, I am a free man”
A. Smith, S. Harris
The Prisoner, Number of the Beast
Iron Maiden, 1982

Índice

Resumen

La inyección de fallos es una técnica utilizada para la validación experimental de Sistemas Tolerantes a Fallos. Se distinguen tres grandes categorías: inyección de fallos física (denominada también *physical fault injection* o *hardware implemented fault injection*), inyección de fallos implementada por *software* (en inglés *software implemented fault injection*) e inyección de fallos basada en simulación. Una de las que más auge está teniendo últimamente es la inyección de fallos basada en simulación, y en particular la inyección de fallos basada en VHDL. Las razones del uso de este lenguaje se pueden resumir en:

- Es un lenguaje estándar ampliamente utilizado en el diseño digital actual.
- Permite describir el sistema en distintos niveles de abstracción.
- Algunos elementos de su semántica pueden ser utilizados en la inyección de fallos.

Para realizar la inyección de fallos basada en VHDL, diferentes autores han propuesto tres tipos de técnicas. La primera está basada en la utilización de los comandos del simulador para modificar los valores de las señales y variables del modelo. La segunda se basa en la modificación del código, insertando perturbadores en el modelo o creando mutantes de componentes ya existentes. La tercera técnica se basa en la ampliación de los tipos del lenguaje y en la modificación de las funciones del simulador VHDL. Actualmente, ha surgido otra tendencia de la inyección de fallos basada en VHDL, denominada genéricamente emulación de fallos. La emulación añade ciertos componentes al modelo (inyectores, que suelen ser perturbadores o mutantes, disparadores de la inyección, recolectores de datos, etc.). El modelo junto con los nuevos componentes son sintetizados en una FPGA, que es donde se realiza la inyección.

Con la introducción cada vez mayor de sistemas tolerantes a fallos en aplicaciones críticas, su validación se está convirtiendo en uno de los puntos clave para su uso. Un ejemplo se da en el campo de la aviación y de la automoción, donde la introducción del concepto *x-by-wire* implica la sustitución de ciertas partes mecánicas por componentes electrónicos. Los grandes requerimientos de seguridad, fiabilidad, etc. necesarios para la introducción de componentes electrónicos en aviación y automoción implica la validación de los circuitos necesarios para la implementación del concepto *x-by-wire*.

Así pues, la validación de sistemas tolerantes a fallos para aplicaciones críticas mediante la aplicación de la inyección de fallos basada en VHDL es una de las tareas pendientes, a pesar de que los lenguajes de descripción de *hardware* en general, y el VHDL en particular, se utilizan cada vez más durante la fase de diseño de los circuitos integrados. Una prueba de su importancia es la continua financiación que la Unión Europea proporciona a la validación de estos sistemas.

A partir de todos estos datos, en la presente tesis se han estudiado, implementado, y en algunos casos mejorado diferentes técnicas de inyección de fallos basadas en VHDL. Una vez desarrolladas las diferentes técnicas, se ha validado el modelo de un sistema tolerante a fallos en tiempo real para aplicaciones críticas. Este modelo se está sintetizando y aplicando en la industria de aviación, así como en la industria de automoción.

Abstract

Fault injection is a technique used for the experimental validation of fault tolerant systems. This technique can be classified in three great categories: physical fault injection, called also hardware implemented fault injection, software implemented fault injection and simulation-based fault injection. Nowadays, one of the most expansive simulation-based fault injection techniques is the VHDL-based fault injection. The reasons of the use of this language can be summarised in:

- It is a standard language broadly used in the current digital design.
- It allows describing the system in different levels of abstraction.
- Some elements of their semantics can be used in the fault injection.

To perform the VHDL-based fault injection, different authors have proposed three types of techniques. The first one is based on the use of the simulator commands to modify the values of the signals and variables of the model. The second one is based on the modification of the code, inserting saboteurs or creating mutants of already existing components. The third technique is based on the amplification of language types and the modification of the VHDL simulator functions. Currently, another tendency of VHDL-based fault injection has arisen, generically denominated fault emulation. This emulation adds certain components to the model (fault injectors, that are usually saboteurs or mutants, injection triggers, data collectors, etc.). The model together with the new components is synthesised in a FPGA that is where the fault injection is performed.

With the continuos introduction of fault tolerant systems in critical applications, their validation is becoming one of the key points for its use. An example is given in the field of aviation and automotive industries, where the introduction of the concept *x-by-wire* implies the substitution of certain mechanical parts for electronic components. The great requirements of security, reliability, etc. needed for the introduction of electronic components in aviation and automotive industries implies the validation of the necessary circuits for the implementation of the *x-by-wire* concept.

Therefore, the validation of fault tolerant systems for critical applications by means of the application of the VHDL-based fault injection is one of the pending tasks, although the hardware description languages in general, and the VHDL in particular, are used more and more during the design phase of integrated circuits. An evidence of its importance is the continuous financing that the European Union provides to the validation of these systems.

From all these data, this thesis presents the study, implementation, and in some cases, the improvement of different techniques of VHDL-based fault injection. Once developed the different techniques, a model of a fault tolerant distributed real time system for critical applications has been validated. This model is synthesising and applying in the aviation industry, as well as in the automotive industry.

Resum

La injecció de fallades és una tècnica utilitzada per a la validació experimental de sistemes tolerants a fallades. Es distingixen tres grans categories: injecció de fallades físiques (denominada també *physical fault injection* o *hardware implemented fault injection*), injecció de fallades implementades per *software* (en anglés *software implemented fault injection*) i injecció de fallades basada en simulació. Una de les què més projecció està tenint últimament és la injecció de fallades basada en simulació, i en particular, la injecció de fallades en models VHDL. Les raons de l'ús d'aquest llenguatge es poden resumir en:

- És un llenguatge estàndard àmpliament utilitzat en el disseny digital actual.
- Permet descriure el sistema en distints nivells d'abstracció.
- Alguns elements de la seua semàntica poden ser utilitzats en la injecció de fallades.

Per realitzar la injecció de fallades basada en VHDL, diferents autors han proposat tres tipus principals de tècniques. La primera està basada en la utilització dels comandaments del simulador per modificar els valors dels senyals i variables del model. La segona està basada en la modificació del codi, inserint pertorbadors en el model o creant mutants de components ja existents. La tercera tècnica es basa en l'ampliació dels tipus del llenguatge i en la modificació de les funcions del simulador VHDL. Actualment, ha sorgit una altra tendència de la injecció de fallades basades en VHDL, denominada genèricament emulació de fallades. L'emulació afegeix certs components al model (injectors, que solen ser pertorbadors o mutants, disparadors de la injecció, recollidors de dades, etc.) i el model junt amb els nous components són sintetitzats en una FPGA, que és on es realitza la injecció.

Amb la introducció cada vegada major de sistemes tolerants a fallades en aplicacions crítiques, la seua validació s'està convertint en un dels punts clau per al seu ús. Un exemple es dóna en el camp de l'aviació i de l'automoció, on la introducció del concepte *x-by-wire* implica la substitució de certes parts mecàniques per components electrònics. Els grans requeriments de seguretat, fiabilitat, etc. necessaris per a la introducció de components electrònics en aviació i automoció implica la validació dels circuits necessaris per a la implementació del concepte *x-by-wire*.

Així doncs, la validació de sistemes tolerants a fallades per a aplicacions crítiques per mitjà de l'aplicació de la injecció de fallades basada en VHDL és una de les tasques pendents, encara que els llenguatges de descripció de *hardware* en general, i el VHDL en particular, s'utilitzen cada vegada més durant la fase de disseny dels circuits integrats. Una prova de la seua importància és el continu finançament que la Unió Europea proporciona a la validació d'aquestos sistemes.

A partir de totes aquestes dades, en la present tesi s'han estudiat, implementat, i en alguns casos millorat diferents tècniques d'injecció de fallades basades en VHDL. Una vegada desenvolupades les diferents tècniques, s'ha validat el model d'un sistema tolerant a fallades en temps real per a aplicacions crítiques. Aquest model s'està sintetitzant i aplicant en la indústria d'aviació, així com en la indústria d'automoció.

1 Presentación

1.1 Fundamentos y motivación

De todos es conocido que, actualmente, los sistemas informáticos ocupan un papel cada vez más importante en nuestra vida cotidiana. Desde computadores personales a pequeños electrodomésticos que se usan a diario, pasando por la gestión de los sistemas de transporte, sistemas bancarios o complejos sistemas industriales o de instalaciones civiles (plantas de energía, presas, etc.). Por esto, el conseguir que los sistemas informáticos duren el mayor tiempo posible y con un elevado nivel de eficacia se ha vuelto una prioridad esencial, en especial, aquellos sistemas en los que un mal funcionamiento puedan causar grandes pérdidas económicas e incluso de vidas humanas.

Los **Sistemas Tolerantes a Fallos** son sistemas que disponen de mecanismos especiales, los cuales proporcionan una cierta inmunidad a la ocurrencia de averías que puedan causar un cese o deterioro del servicio prestado. Se denomina **Confiabilidad** al conjunto de funciones (denominados también atributos) que permiten cuantificar la calidad del servicio prestado y el grado de confianza que el usuario puede depositar en el sistema. Esta serie de atributos son dependientes del punto de vista utilizado para determinar la calidad del servicio, como por ejemplo la **Fiabilidad** o la **Seguridad-Inocuidad**.

La **validación** o valoración de la calidad del servicio que presta un sistema se puede realizar de forma **teórica** o **experimental**. La validación teórica se realiza resolviendo un modelo teórico del sistema, mientras que la validación experimental se basa en observar el comportamiento ante fallos del sistema real o de un modelo del mismo. Ciertos parámetros del sistema, como por ejemplo los coeficientes de cobertura en la detección y/o en la recuperación de errores o los tiempos de latencia en la propagación de los errores, son necesarios para la validación teórica. Sin embargo, estos parámetros no son fácilmente estimables de forma teórica.

La validación experimental de un sistema puede realizarse mediante la **observación del sistema** durante su fase operativa o mediante la **inyección de fallos**. El primer método tiene como principal inconveniente la baja ocurrencia de fallos en sistemas tolerantes a fallos. En cambio, con el segundo método se provocan fallos de forma deliberada en el sistema bajo estudio, por lo que se facilita la observación del sistema cuando ocurren averías, sirviendo también como apoyo a la validación teórica, ya que se aporta al modelo teórico los parámetros necesarios.

Existen diferentes técnicas para realizar la inyección de fallos, que se pueden clasificar en tres grandes grupos: inyección física (o HWIFI, del inglés *hardware implemented fault injection*), inyección implementada por *software* (o SWIFI, del inglés *software implemented fault injection*), e inyección mediante simulación.

Una de las líneas maestras de investigación del Grupo de Sistemas Tolerantes a Fallos (GSTF) del Departamento de Informática de Sistemas y Computadores de la Universidad Politécnica de Valencia, al que el autor de esta tesis doctoral pertenece, es el desarrollo de herramientas de inyección de fallos. Se han desarrollado herramientas en las tres áreas: HWIFI [PGil92, PGil97, Martínez99, PGil03], SWIFI [Campelo99, Yuste03] y simulación [Baraza02, Baraza03, DGil03b].

En la actualidad, para el diseño y simulación de sistemas digitales se utilizan unos lenguajes especiales, llamados lenguajes de descripción de *hardware* (en inglés *hardware description languages*, o HDL). Uno de los más extendidos es el VHDL (*Very high speed integrated circuits Hardware Description Language*), que ofrece gran versatilidad para el diseño y simulación de sistemas a diferentes niveles, desde una puerta a un sistema completo. Además su

semántica ofrece propiedades que lo hacen muy adecuado para inyectar fallos. Es por estas razones que se ha elegido la inyección de fallos basada en VHDL para realizar la presente tesis.

1.2 Objetivos

Los principales objetivos de esta tesis se pueden dividir en diferentes tareas que se complementan entre sí. Partiendo de un exhaustivo estudio de la inyección de fallos como un método general para la validación de la Confiabilidad de los sistemas tolerantes a fallos, se incidirá de manera especial en la inyección de fallos basada en VHDL, y se realizará un estudio de las diferentes técnicas de inyección basadas en VHDL presentadas en la literatura, para después implementar y, sobretodo, mejorar dichas técnicas.

En particular, se estudiarán tres técnicas: órdenes del simulador, perturbadores y mutantes, las cuales presentan diferentes características en cuanto a facilidad de implementación, de automatización, modelos de fallos, etc.

La técnica de las órdenes del simulador se basa en la utilización de ciertas órdenes del simulador para controlar la simulación del modelo, alterando el valor y/o la temporización de los elementos internos del mismo. Esta técnica es la más sencilla de implementar y automatizar, siendo además la más rápida de ejecutar. Su principal inconveniente es su total dependencia con el simulador y el reducido conjunto de modelos de fallos que puede aplicar.

Un perturbador se puede definir como un componente que se añade al modelo VHDL, y cuya función es alterar el valor o las características temporales de una o más señales a la hora de inyectar un fallo. Su principal ventaja es la ampliación de los modelos de fallos que se pueden inyectar, aunque presenta cierta complejidad de implementación y automatización. En cuanto al tiempo de simulación, éste es mayor que el tiempo empleado con la técnica de las órdenes del simulador, debido principalmente a la simulación del modelo más la simulación de los perturbadores añadidos al mismo.

Un mutante es un nuevo componente VHDL que reemplaza al componente original, comportándose como el componente al que reemplaza durante su funcionamiento normal, e inyectando el fallo cuando el componente mutado simula el comportamiento del componente original en presencia de fallos. La ventaja principal de esta técnica es su gran variedad de modelos de fallos, muy superior a los de las otras dos. Su mayor problema es el gran consumo temporal de las simulaciones, sobretodo a la hora de inyectar fallos transitorios.

Para poder probar las diferentes alternativas que se implementarán, éstas se aplicarán sobre un microprocesador tolerante a fallos, en el que se realizarán una serie de experimentos de inyección de fallos con el fin de calcular diferentes parámetros de la Confiabilidad, así como para estudiar la respuesta del sistema en presencia de diferentes modelos de fallos, tanto transitorios como permanentes.

Estos experimentos de inyección de fallos permitirán estudiar la viabilidad de las diferentes técnicas de inyección de fallos así como sus posibles mejoras. Además, la utilización de estas técnicas implica también el estudio y la aplicación de nuevos modelos de fallos, que amplíen los modelos de fallos utilizados hasta ahora, como eran el *bit-flip* para fallos transitorios y el *stuck-at* para fallos permanentes.

Para poder realizar los distintos experimentos de inyección de fallos, se utilizará VFIT, una herramienta de inyección de fallos desarrollada por el GSTF. Un objetivo de la presente tesis será la implementación y mejora del módulo analizador de VFIT, mejorando sus prestaciones a medida que los modelos VHDL que se utilizarán durante los diferentes experimentos de inyección de fallos se vuelvan más complejos, ampliándose también los posibles parámetros que se puedan obtener durante el análisis con VFIT.

El siguiente paso en el desarrollo de la presente tesis consistirá en la validación de un sistema distribuido de tiempo real tolerante a fallos. Todos los experimentos de inyección de fallos sobre este sistema se han llevado a cabo durante el proyecto europeo denominado "*FIT:*

Fault Injection in the Time-Triggered Architecture" (IST-1999-10748). En este proyecto se ha validado el controlador de comunicaciones *TTP/C* mediante el uso de diferentes técnicas de inyección. Las técnicas usadas abarcan los tres grandes tipos de técnicas de inyección de fallos: inyección física, inyección implementada por *software* e inyección mediante simulación.

Bajo el objetivo global de comprobar si el sistema bajo prueba cumple la propiedad de silencio ante fallos (el sistema transmite mensajes correctamente o deja de transmitir mensajes en presencia de fallos), se realizarán una serie de experimentos de inyección de fallos que además servirán para cumplir diferentes objetivos tanto de la presente tesis en particular como del proyecto *FIT* en general.

El modelo que se empleará en este segundo bloque de experimentos ha sido utilizado por los diseñadores del circuito integrado durante la síntesis del controlador de comunicaciones *TTP/C*, un microprocesador tolerante a fallos basado en la arquitectura *Time-Triggered*. Este sistema integra un protocolo de comunicaciones para sistemas distribuidos de tiempo real tolerantes a fallos. Los experimentos de inyección se realizarán en dos modelos de dos versiones diferentes: el *TTP/C-C1* y el *TTP/C-C2*, siendo el *TTP/C-C1* el primer controlador de la familia *TTP/C* (primer controlador de comunicaciones basado en la arquitectura *Time-Triggered*). Estos controladores se están aplicando actualmente en campos como la aviónica y la automoción, siendo utilizados como sustitutos de los controladores de ciertas partes mecánicas, implementando el concepto *x-by-wire*.

En primer lugar, se realizarán una serie de experimentos de inyección de fallos que servirán para la determinación de la cobertura de detección de errores de los diferentes mecanismos de detección de errores del controlador *TTP/C*. Estos experimentos también servirán para la identificación de debilidades en el modelo en VHDL de ambos controladores. A partir de los experimentos anteriores, se realizarán diferentes propuestas para corregir o mejorar dicho modelo VHDL.

El siguiente paso será la comparación y/o combinación con otras técnicas de inyección de fallos. En este caso, el trabajo se enfocará por dos caminos distintos. En primer lugar, se hará una comparación de las diferentes técnicas de inyección de fallos que se han utilizado durante el proyecto *FIT*. En segundo lugar, se realizarán una serie de experimentos de inyección de fallos en colaboración con la técnica de inyección de fallos a nivel de *pin*. Con esta serie de experimentos se podrán identificar nuevas debilidades tanto del modelo VHDL como del prototipo del controlador, proponiéndose las mejoras correspondientes.

1.3 Desarrollo

Esta tesis recoge el trabajo realizado para la consecución de los objetivos anteriormente expuestos, y describe los resultados obtenidos. A continuación se especifica la organización de esta tesis.

El capítulo 2, titulado "Generalidades de los Sistemas Tolerantes a Fallos", está dedicado a la terminología utilizada en el campo de la Tolerancia a Fallos. En él se establece la relación entre fallo, error y avería, y se define el concepto de Confiabilidad y sus atributos. En este capítulo también se enumeran los diferentes métodos existentes para evaluar la Confiabilidad de un sistema, haciendo especial hincapié en la validación de forma experimental, en particular en la validación mediante inyección de fallos, donde se hace un estudio exhaustivo del *estado del arte* en este campo, explicando las diferentes técnicas y herramientas existentes, con excepción de la inyección de fallos sobre modelos en VHDL, a las que se dedica un capítulo aparte.

En el capítulo 3, "Técnicas de inyección de fallos basadas en VHDL", se describe el *estado del arte* de la inyección de fallos basada en VHDL, describiendo las distintas variantes existentes y enumerando las herramientas de inyección más relevantes. En este capítulo también se describen detalladamente las distintas subtécnicas de inyección de fallos basada en VHDL que se han estudiado e implementado a lo largo de la presente tesis, resaltando en cada una de ellas sus posibles mejoras así como los modelos de fallos que permiten aplicar.

En el capítulo 4, "VFIT: La herramienta de inyección de fallos. Modelos de fallos", se hace una breve descripción de VFIT, la herramienta de inyección de fallos en modelos VHDL que ha sido desarrollada en el GSTF. También se resumen los mecanismos de fallo que más importancia tienen (o se prevé que van a tener) en los circuitos integrados submicrónicos actuales, deduciéndose de este estudio un conjunto de modelos de fallos (transitorios, permanentes e intermitentes) que serán los que se aplicarán en los experimentos de inyección. Hay que mencionar que estos dos temas se han tratado de una forma muy resumida, ya que no es el objetivo principal de esta tesis. Sin embargo, se ha incluido un resumen de ambos temas debido a su extensivo uso durante el desarrollo de la presente tesis.

Una vez descritas las técnicas de inyección de fallos, la herramienta que se va a utilizar durante los diferentes experimentos de inyección y los modelos de fallos a inyectar, el capítulo 5, "Aplicación de nuevas técnicas de inyección de fallos en modelos VHDL", describe los distintos experimentos de inyección realizados. Para cada experimento se especifican los objetivos buscados, los parámetros necesarios para realizar la inyección y las técnicas de inyección utilizadas, y se muestran los resultados extraídos, en forma de tablas y gráficos, indicando los valores de latencias, coberturas, etc. calculados, describiendo de esta forma los resultados más interesantes obtenidos de la inyección de fallos.

Una vez que se han probado las diferentes técnicas de inyección en modelos VHDL, se ha seleccionado una de ellas para su aplicación sobre un modelo real de un sistema tolerante a fallos en tiempo real para aplicaciones críticas. En primer lugar, el capítulo 6, "Arquitecturas de bus para sistemas distribuidos de tiempo real tolerantes a fallos. Introducción a la arquitectura Time-Triggered", realiza un pequeño estudio del *estado del arte* de este tipo de sistemas, así como de su validación, para después describir la arquitectura *Time-Triggered* y el controlador *TTP/C*, basado en esta arquitectura, que actualmente se están utilizando en la industria de automoción y aviación. Así pues, una vez que la arquitectura *Time-Triggered* y el controlador *TTP/C* han sido descritos, el capítulo 7, "Validación de sistemas distribuidos de tiempo real tolerantes a fallos para aplicaciones críticas", introduce la problemática de su validación. Para ello, primero realiza un estudio de los diferentes métodos utilizados para validar sistemas tolerantes a fallos en tiempo real para aplicaciones críticas, así como de los diferentes proyectos europeos cuya tarea principal ha sido la validación de este tipo de sistemas. La segunda parte de este capítulo describe detalladamente los diferentes experimentos de inyección de fallos realizados durante la validación del modelo del controlador *TTP/C*. Estos experimentos se han dividido en dos grupos. El primero muestra la validación del modelo en VHDL mediante el

estudio del funcionamiento de los mecanismos de detección de errores del controlador en presencia de fallos transitorios, mientras que el segundo grupo de experimentos muestra la comparación y/o combinación de la inyección de fallos basada en VHDL con la inyección de fallos a nivel de *pin*.

Para finalizar, el capítulo 8, "Conclusiones. Trabajo futuro", presenta una recapitulación del trabajo presentado, y se extraen las conclusiones más interesantes a las que se ha llegado durante el desarrollo de la presente tesis, así como se explican las líneas de investigación abiertas con esta tesis.

2 Generalidades de los Sistemas Tolerantes a Fallos

Como primer paso de la presente tesis, en este capítulo se va a introducir la terminología y los conceptos básicos que se utilizarán a lo largo del desarrollo de la misma. Todo este glosario de términos está basado en los trabajos presentados en [LIS96, PGil96, Gracia00b, Baraza03 y Yu03].

2.1 Generalidades de los Sistemas Tolerantes a Fallos

2.1.1 Introducción

Normalmente, todos los sistemas presentan una cierta tolerancia a fallos debida a redundancias internas no intencionadas así como a una cierta inmunidad inherente en dicho sistema. Los sistemas tolerantes a fallos integran además mecanismos especiales diseñados específicamente para el tratamiento de estas situaciones. Para que el usuario pueda cuantificar y/o incrementar la confianza en un sistema, se debe llevar a cabo una evaluación de éste, sobretodo si se pretende que realice alguna tarea crítica. Se denomina **Confiabilidad**[1] a la propiedad de un sistema informático que permite depositar una confianza justificada en el servicio que proporciona. Este servicio proporcionado por un sistema es el comportamiento percibido por sus usuarios, siendo este usuario otro sistema (físico o humano) que interactúa con el primero. Para conseguir que un sistema tenga una gran Confiabilidad, se deben tener en cuenta tres puntos durante el proceso de diseño del sistema:

1. Especificación de los requerimientos de la Confiabilidad del sistema.
2. Diseño e implementación del sistema, de tal manera que alcance la Confiabilidad requerida en función de la especificación.
3. Validación del sistema, para asegurar que se ha alcanzado una cierta Confiabilidad.

La Confiabilidad presenta una serie de atributos:

- **Disponibilidad** (en inglés, *Availability*): Preparación para el servicio.
- **Fiabilidad** (en inglés, *Reliability*): Continuidad del servicio.
- **Seguridad**–**Inocuidad** (en inglés, *Safety*): Ausencia de consecuencias catastróficas en el entorno.
- **Confidencialidad** (en inglés, *Confidentiality*): Ausencia de revelaciones de información no autorizadas.
- **Integridad** (en inglés, *Integrity*): Ausencia de alteraciones indebidas de información.
- **Mantenibilidad** (en inglés, *Maintainability*): Capacidad para someterse a reparaciones y evoluciones.

Sin embargo, existen una serie de impedimentos de la Confiabilidad que provocan que el servicio proporcionado por el sistema no cumpla con la función del mismo, siendo la función del sistema la tarea que debe realizar. Se dice que se produce una avería en un sistema cuando el servicio proporcionado incumple la función del sistema. Un error es la parte del estado del sistema responsable de llevar a éste a una avería. Un error que afecta al servicio es una indicación de que la avería está ocurriendo o ha ocurrido. Un fallo es la causa justificada o

[1] Del inglés, *Dependability*. Anteriormente se la conocía como Garantía de funcionamiento [PGil96].

hipotética de un error. Es decir, un error es la manifestación de la existencia de un fallo en el sistema, y una avería es el efecto que el error produce en el servicio del sistema.

El desarrollo de sistemas confiables[2] (esto es, con una elevada Confiabilidad) requiere la utilización combinada de un conjunto de métodos que pueden clasificarse en:

- **Prevención de fallos**: cómo prevenir la aparición de fallos.
- **Tolerancia a fallos**: cómo proporcionar un servicio que cumpla la función del sistema a pesar de los fallos.
- **Eliminación de fallos**: cómo reducir la presencia (número y gravedad) de los fallos.
- **Predicción de fallos**: cómo estimar el número actual de fallos, así como su incidencia futura y sus consecuencias.

Para conseguir una alta Confiabilidad, se utilizan la prevención y la tolerancia a fallos. Con estos métodos el sistema es capaz de proporcionar el servicio esperado. Si lo que se pretende es realizar una validación del sistema, es necesario emplear la eliminación y la predicción de fallos, con las cuales se cuantifica y justifica la confianza depositada en que el sistema sea capaz de proporcionar el servicio requerido.

Todos estos conceptos se pueden ver esquematizados en la Figura 1, donde se muestra la Confiabilidad de un sistema como un conjunto compuesto por:

- **Atributos.** Expresan las propiedades que se esperan de un sistema y valoran la calidad del servicio proporcionado.
- **Impedimentos.** Circunstancias no deseadas pero no por ello inesperadas que causan o son el resultado de una falta de Confiabilidad.
- **Medios.** Técnicas y métodos utilizados para conseguir la Confiabilidad:
  1. Proporcionan al sistema la capacidad para entregar un servicio en el que se pueda confiar.
  2. Evalúan y cuantifican la confianza en esta capacidad.

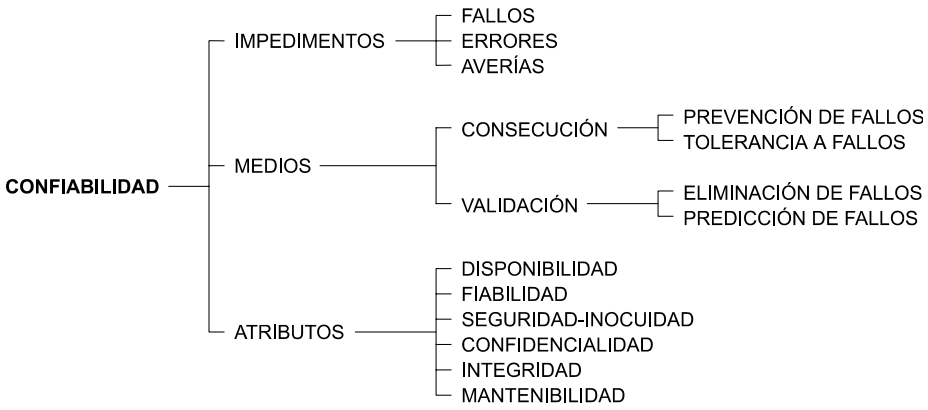


Figura 1. Arbol de la Confiabilidad.

De este modo, la Confiabilidad es el resultado de la interacción entre los impedimentos y los medios que se oponen a éstos sobre los atributos fijados.

En los siguientes apartados se explicará un poco más detalladamente los componentes del árbol de la Figura 1, con el fin de justificar la necesidad de evaluar (validar) un sistema para así poder cuantificar la confianza que sus usuarios pueden tener en el servicio proporcionado.

[2] Del inglés, *Dependable systems*.

2.1.2 Atributos de la Confiabilidad

Según las definiciones del apartado anterior, la disponibilidad y la fiabilidad están relacionadas con la capacidad de evitar las averías, mientras que la seguridad–inocuidad da una idea de la capacidad de evitar averías catastróficas Algunas definiciones más formales de los atributos de la Confiabilidad serían [PGil92]:

- **Fiabilidad, R(t)**. Es la probabilidad condicional de que el sistema, funcionando correctamente en el instante t_0, esté funcionando sin interrupciones durante el intervalo $[t_0, t]$, con $t > t_0$.
- **Disponibilidad, A(t)**. Es la probabilidad de que el sistema funcione correctamente en el instante t.
- **Mantenibilidad, M(t)**. Similar a la fiabilidad, pero se refiere al tiempo transcurrido entre el estado de servicio inadecuado y el de servicio adecuado.
- **Seguridad–Inocuidad, S(t)**. Es la probabilidad condicional de que el sistema, estando funcionando correctamente en un instante t_0, continúe funcionando sin interrupciones, o se encuentre en un estado de avería no catastrófica durante el intervalo $[t_0,t]$, con $t > t_0$.

2.1.3 Impedimentos de la Confiabilidad

En este apartado se van a estudiar aquellas circunstancias, que aunque no son deseadas, tampoco son inesperadas, y que provocan la no Confiabilidad en el sistema. En concreto, se van a examinar los conceptos de avería, error y fallo, así como sus mecanismos de manifestación, es decir, la patología de los fallos

2.1.3.1 Averías

En el apartado 2.1.1 se ha definido el comportamiento de una avería como un incumplimiento de la función del sistema, no como un incumplimiento de su especificación. Sin embargo, hay que tener en cuenta que un comportamiento correcto desde el punto de vista de la especificación puede ser un comportamiento inaceptable desde el punto de vista de los usuarios del sistema, dejando al descubierto un fallo de especificación.

Las formas en que un sistema puede averiarse se denominan modos de avería, los cuales se pueden describir desde tres puntos de vista: dominio, percepción por los usuarios del sistema y consecuencias en el entorno. Desde el punto de vista del *dominio* de la avería se puede distinguir entre:

- **Averías de valor**: el valor del servicio entregado no cumple con la función del sistema.
- **Averías de tiempo**: el tiempo del servicio entregado no cumple con la función del sistema. Éstas se subdividen, a su vez, en averías con adelanto y con retraso, dependiendo de si el servicio se ha entregado demasiado pronto o demasiado tarde.

Una clase de averías relacionadas a la vez con el valor y el tiempo son las denominadas averías con parada, que ocurren cuando la actividad del sistema (si la hay) no es perceptible por sus usuarios. En función de cómo el sistema interactúa con sus usuarios, tal ausencia de actividad puede tomar la forma de:

- **Salidas congeladas**, en el que se entrega un servicio de valor constante.
- **Silencio**, cuando no se entrega nada.

Un sistema cuyas averías son solamente con parada se denomina, de manera general, sistema con parada tras avería (del inglés, *Fail-stop)*. Las situaciones de salidas congeladas o en silencio dan lugar, respectivamente, a sistemas pasivos tras avería (del inglés, *Fail-passive*) y a sistemas en silencio tras avería (del inglés, *Fail-silent*).

El punto de vista de la percepción de la avería lleva a distinguir, en caso de que haya varios usuarios, entre:

- **Averías coherentes**: todos los usuarios del sistema tienen la misma percepción de todas las averías.
- **Averías incoherentes**: los usuarios del sistema pueden percibir alguna avería de forma diferente, denominadas comúnmente **averías bizantinas**.

En relación con la clasificación realizada desde el punto de vista del dominio, hay que resaltar que las averías de un sistema en silencio tras avería son coherentes, mientras que pueden no serlo en un sistema pasivo tras avería.

La clasificación de las consecuencias sobre el entorno del sistema se lleva a cabo estableciendo un criterio para la gravedad de las averías. El número, la denominación y la definición de los niveles de gravedad, así como las probabilidades de ocurrencia admisibles dependen normalmente de las aplicaciones. Sin embargo, pueden definirse dos niveles extremos:

- **Averías benignas**. Las consecuencias son de un orden de magnitud igual al beneficio obtenido por el servicio entregado en ausencia de averías.
- **Averías catastróficas**. Las consecuencias son muy superiores al beneficio obtenido por el servicio entregado en ausencia de averías.

La Figura 2 resume las anteriores clasificaciones de las averías.

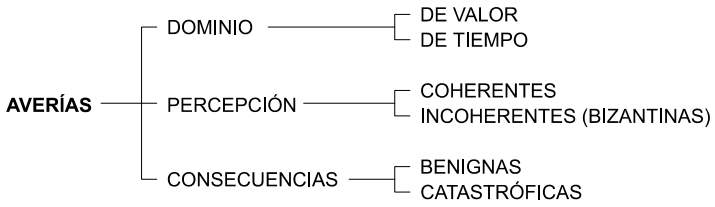


Figura 2. Clases de averías.

Un sistema donde todas las averías son (en una medida aceptable) benignas se denomina sistema seguro tras avería[3]. La noción de gravedad de las averías permite definir el concepto de la **criticidad** de un sistema como la mayor gravedad de sus posibles modos de avería. La relación existente entre los modos de avería y la gravedad de las averías es muy dependiente de la aplicación considerada. Sin embargo, existe una amplia clase de aplicaciones donde la inactividad se considera como una posición segura por naturaleza (por ejemplo, transportes terrestres, producción de energía, etc.), de donde se deduce la correspondencia directa que existe a menudo entre parada tras avería y seguridad en presencia de averías.

Los sistemas con parada tras avería (pasivos o en silencio) y los sistemas seguros tras avería son ejemplos de sistemas controlados tras avería; es decir, sistemas que han sido diseñados y realizados con el fin de que se averíen solamente, o en una medida aceptable, de acuerdo a modos restrictivos de avería; por ejemplo, que tengan las salidas congeladas en vez de entregar valores erráticos, que permanezcan en silencio en lugar de emitir mensajes erróneos, que posean averías coherentes en vez de incoherentes, etc. Los sistemas controlados tras avería pueden ser definidos, además, imponiendo alguna condición al estado interno o a la accesibilidad.

2.1.3.2 Errores

Se ha definido al error como el responsable de provocar una avería, aunque el hecho de que un error conduzca o no a una avería depende de tres factores principales:

[3] Del inglés *Fail-safe*.

1. De la composición del sistema, y en particular, de la naturaleza de la redundancia existente:
   - **Intencionada** o extrínseca, destinada explícitamente a evitar que un error dé lugar a una avería.
   - **No intencionada** o intrínseca, que puede tener el mismo efecto, aunque inesperado, que la intencionada.
2. De la actividad del sistema. Según esta actividad, puede suceder que no se utilice la parte en la que el error está presente, o que el sistema modifique la parte errónea transformándola en correcta. En ambos casos, el error no provocará una avería.
3. De la definición de avería desde el punto de vista del usuario: Lo que para un usuario dado es una avería, puede no ser más que una molestia soportable para otro.

2.1.3.3 Fallos

Los fallos se pueden clasificar de acuerdo a cinco puntos de vista principales:

- Sus causas fenomenológicas, que distinguen los fallos entre **fallos físicos** y **fallos humanos**.
- Su naturaleza, que diferencia entre **fallos accidentales** y **fallos intencionados**.
- La fase de vida del sistema en el que ocurren, que distingue entre **fallos de desarrollo** y **fallos de operación**.
- Su situación respecto a las fronteras del sistema. En este caso, los fallos se pueden clasificar en **internos** y **externos**.
- Su persistencia (modo de manifestarse en el tiempo):
  - ⇨ **Permanentes**, cuya presencia no está ligada a condiciones puntuales, sean **internas** o **externas**.
  - ⇨ **Temporales**, cuya presencia está ligada a dichas condiciones y están presentes durante un tiempo limitado. De éstos, a los **fallos temporales externos** se les denomina **transitorios**, mientras que a los **fallos temporales internos** se les llama **intermitentes**, y son el resultado de la presencia de combinaciones o condiciones que se dan raramente.

La Figura 3 resume las diferentes clases de fallos recién expuestos según los diferentes puntos de vista considerados. Estos fallos pueden denominarse elementales. En la práctica, dependiendo del punto de vista con que se mire, un fallo determinado puede enclavarse en varias de estas clases elementales. Obviamente no todas las combinaciones son posibles, ya que algunas clases son excluyentes entre sí. En la Figura 4 se muestran las combinaciones válidas que se pueden producir, las cuales dan lugar a cinco categorías, los **fallos físicos** propiamente dichos y cuatro categorías de **fallos humanos**:

- **Fallos de diseño**: incluye los fallos de desarrollo, accidentales o intencionados no malévolos.
- **Fallos de interacción**: fallos externos de operación, accidentales o intencionados no malévolos.
- **Lógica malévola**: fallos internos, intencionados malévolos.
- **Intrusiones**: fallos externos de operación.

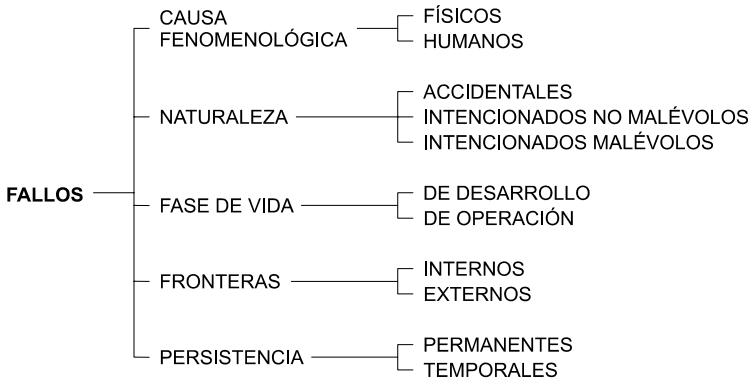


Figura 3. Clases de fallos elementales.

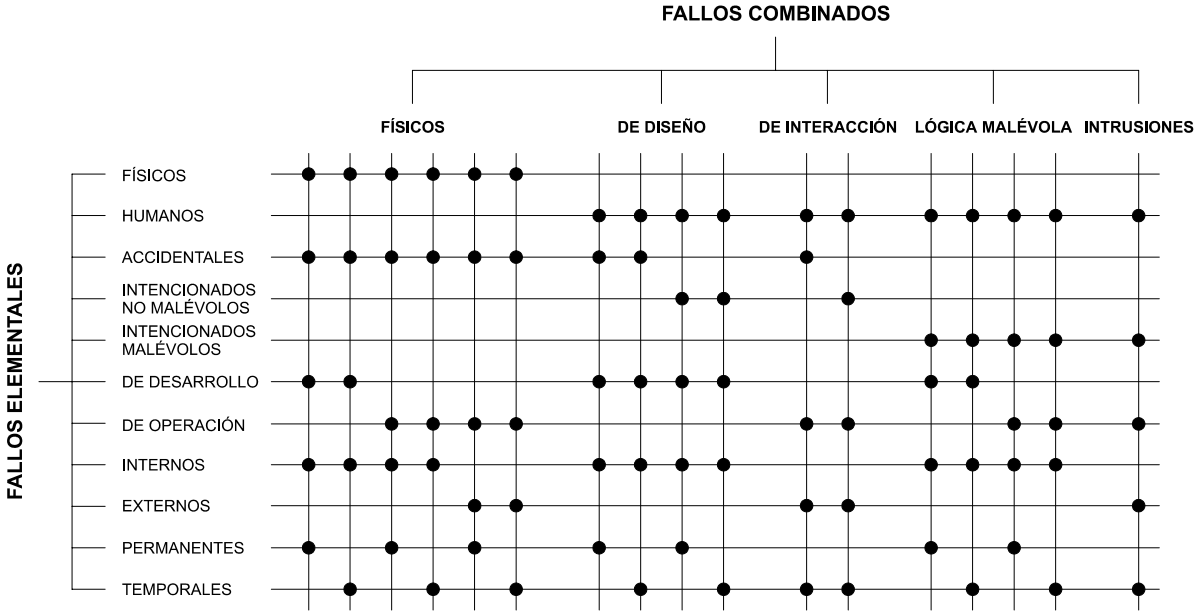


Figura 4. Clases de fallos combinados.

2.1.3.4 Patología de los fallos

Un fallo es **activo** cuando produce un error. Un fallo activo puede ser un fallo externo, o uno interno que se encontraba previamente **dormido**, y que ha sido activado por el proceso de computación. Un error puede ser **latente** o **detectado**. Un error es latente cuando no se ha reconocido como tal; un error es detectado por un algoritmo o un mecanismo de detección. Un error puede desaparecer antes de ser detectado. Un error puede propagarse, y generalmente lo hace. Mediante la propagación, un error crea otros (nuevos) errores. Durante la operación del sistema, la presencia de fallos activos sólo puede ser determinada mediante la detección de errores. Una avería ocurre cuando un error atraviesa la frontera sistema–usuario y afecta al servicio entregado por el sistema. Estos mecanismos permiten completar la siguiente *cadena fundamental*:

Fallo ➡ Error ➡ Avería ➡ Fallo ➡ ...

La transformación entre los estados de fallo, error y avería no se produce de forma instantánea en el tiempo. Así, desde que se produce el fallo hasta que se manifiesta el error existe un tiempo de inactividad, llamado **latencia del error**. Durante este tiempo se dice que el fallo *no es efectivo* y que el error está *latente*. De forma análoga se puede definir la **latencia de la detección del error** y la **latencia de producción de la avería**.

2.1.4 Medios para alcanzar la Confiabilidad

Si recordamos la Figura 1, los medios para alcanzar la Confiabilidad se pueden dividir en medios que proporcionan al sistema la capacidad para entregar un servicio en el que se pueda confiar y medios que evalúan y cuantifican la confianza en esta capacidad. A continuación se explicarán estos medios, excepto la prevención de fallos, ya que por pertenecer a la ingeniería general de los sistemas, requiere de unas técnicas de diseño muy particulares, totalmente diferentes conceptualmente a las demás aquí expuestas.

2.1.4.1 Tolerancia a fallos

La Tolerancia a fallos se puede llevar a cabo mediante:

- El procesamiento de los errores, destinado a eliminarlos del estado computacional, a ser posible antes de que ocurra una avería. Se puede realizar por medio de tres principios de diseño:
  - ⇨ Detección de errores, que permite identificar como tal a un estado erróneo.
  - ⇨ Diagnóstico de errores, que permite apreciar los daños producidos por el error detectado, o por los propagados antes de la detección.
  - ⇨ Recuperación de errores, donde se sustituye el estado erróneo por otro libre de errores. Esta sustitución puede hacerse de tres maneras:
    a) Mediante recuperación hacia atrás. El estado erróneo se sustituye por otro ya sucedido antes de la ocurrencia del error. Este método incluye el establecimiento de puntos de recuperación (en inglés *checkpointing*), que son instantes durante la ejecución de un proceso donde se salvaguarda el estado, pudiendo éste posteriormente ser restaurado tras la ocurrencia de un error.
    b) Mediante recuperación hacia adelante. La transformación del estado erróneo consiste en encontrar un nuevo estado a partir del cual el sistema pueda seguir funcionando (habitualmente en modo degradado).
    c) Mediante compensación. El estado erróneo contiene la suficiente redundancia como para permitir su transformación en un estado libre de errores.
- El tratamiento de los fallos, destinado a prevenir que se activen fallos de nuevo. El primer paso es el diagnóstico, que consiste en la determinación de las causas de los errores. Posteriormente vienen las acciones destinadas a impedir una nueva activación de los fallos. Esta operación se denomina pasivación de los fallos. Este objetivo se lleva a cabo eliminando del proceso de ejecución posterior los elementos considerados defectuosos. Si el sistema no es capaz de seguir dando el mismo servicio que antes, puede tener lugar una reconfiguración, para que los componentes no averiados del mismo permitan la entrega de un servicio aceptable, aunque sea degradado. Una reconfiguración puede implicar la renuncia a algunas tareas, o una reasignación de éstas entre los componentes no averiados. La pasivación del fallo no será precisa si se estima que el procesamiento del error ha podido eliminar el fallo directamente, o si su probabilidad de reaparición es lo suficientemente pequeña. En caso de no tener que realizarse la pasivación, el fallo se considera blando; si se lleva a cabo la pasivación, el fallo se considera duro.

2.1.4.2 Eliminación de fallos

La Eliminación de fallos está constituida por tres etapas: verificación, diagnóstico y corrección. La verificación consiste en determinar si el sistema satisface unas propiedades, llamadas condiciones de verificación. En caso contrario, deben llevarse a cabo las otras dos etapas: diagnosticar el o los fallos que impidieron que se cumpliesen las condiciones de verificación y, posteriormente, realizar las correcciones necesarias. Después de la corrección, el

proceso debe comenzar de nuevo con el fin de comprobar que la eliminación de fallos no haya tenido consecuencias indeseables. Esta última verificación se denomina de no–regresión. Las condiciones de verificación pueden ser generales (se aplican a una clase de sistema dado, independientes de las especificaciones) o particulares del sistema considerado, deducidas directamente de su especificación.

Las técnicas de verificación pueden clasificarse según si implican o no la activación del sistema. La verificación de un sistema sin su activación real se denomina estática, que puede realizarse en el propio sistema, en forma de análisis estático o pruebas de exactitud, o en un modelo de comportamiento del sistema (basado por ejemplo en redes de *Petri* o autómatas de estados finitos), dando lugar a un análisis del comportamiento.

La verificación de un sistema activado se denomina dinámica. En este caso, las entradas suministradas al sistema pueden ser simbólicas, como en el caso de la ejecución simbólica, o con un determinado valor, como en las pruebas de verificación (denominados comúnmente *tests*). También se debe verificar que el sistema no hace nada más que lo que está especificado.

Se denomina diseño para la verificación al diseño de sistemas de forma que su verificación sea fácil. Este planteamiento está especialmente desarrollado con respecto a los fallos físicos, llamándose en este caso particular diseño para la prueba.

La eliminación (corrección) de fallos durante la vida operativa de un sistema se llama mantenimiento correctivo. Puede ser de dos formas:

- Mantenimiento curativo, destinado a eliminar los fallos que han producido uno o más errores que han sido detectados.
- Mantenimiento preventivo, destinado a eliminar los fallos antes de que produzcan errores. Estos fallos pueden ser:

  a) Fallos físicos que han aparecido después de las últimas acciones de mantenimiento preventivo.

  b) Fallos de diseño que han dado lugar a errores en otros sistemas similares.

Estas definiciones se aplican tanto a los sistemas no tolerantes a fallos como a los tolerantes a fallos. Estos últimos pueden, después de la corrección, ser mantenidos en línea (sin interrupción del servicio), o fuera de línea. Es importante notar, finalmente, que la frontera entre el mantenimiento correctivo (eliminación de fallos) y el tratamiento de los fallos (tolerancia a fallos) es relativamente arbitraria; en particular, se puede considerar al mantenimiento curativo como un medio de tolerancia a fallos.

2.1.4.3 Predicción de fallos

La Predicción de fallos se lleva a cabo realizando una evaluación del comportamiento del sistema respecto a la ocurrencia de los fallos y a su activación. Dicha evaluación tiene dos facetas:

- Cualitativa, destinada en primer lugar a identificar, clasificar y ordenar los modos de avería, y en segundo lugar, a identificar las combinaciones de eventos (averías de componentes o condiciones del entorno), que dan lugar a situaciones no deseadas.
- Cuantitativa, destinada a la cuantificación, en términos de probabilidades, de algunos de los atributos de la Confiabilidad.

La definición de las medidas de la Confiabilidad precisa, en primer lugar, de las nociones de servicio correcto e incorrecto. Si el servicio entregado cumple con la función del sistema, el servicio es correcto. Por el contrario, si el servicio entregado no cumple con la función del sistema, el servicio es incorrecto. Una avería es, por tanto, una transición entre servicio correcto e incorrecto. A la transición entre servicio incorrecto y correcto se denomina restauración. La

cuantificación de la alternancia entre servicio correcto e incorrecto permite definir la Fiabilidad y la Disponibilidad como medidas de la Confiabilidad:

- Fiabilidad: medida de la entrega continua de un servicio correcto, o de manera equivalente, del tiempo hasta la avería.
- Disponibilidad: medida de la entrega de un servicio correcto considerando la alternancia entre servicio correcto y servicio incorrecto.

Habitualmente, también se considera una tercera medida: la Mantenibilidad, que puede definirse como la medida del tiempo de restauración después de la última avería, o de manera equivalente, de la entrega continua de un servicio incorrecto.

La Seguridad–Inocuidad puede ser vista como una extensión de la Fiabilidad. Si se agrupa el estado de servicio correcto con el estado de servicio incorrecto posterior a las averías benignas, dentro de un estado seguro (en ausencia de daños catastróficos), la Seguridad–Inocuidad es entonces una medida de la continuidad del servicio seguro–inocuo, o bien del tiempo hasta la avería catastrófica. La Seguridad–Inocuidad puede ser vista, pues, como la Fiabilidad respecto a las averías catastróficas.

En el caso de los sistemas con múltiples prestaciones, pueden distinguirse diversos servicios, así como diversas formas de entregar el servicio, desde la plena capacidad hasta la parada completa, lo cual puede ser visto como entregas de servicio cada vez menos correctas. La medida combinada de Prestaciones y Confiabilidad se denomina habitualmente Prestabilidad[4].

Los dos principales métodos de la *evaluación cuantitativa*, destinados a la obtención de estimadores cuantificados de las medidas de la Confiabilidad, son el modelado y la prueba (de evaluación). Estos métodos son complementarios, ya que el modelado precisa de datos relativos a los modelos de procesos elementales (avería, mantenimiento, activación del sistema, etc.) que pueden obtenerse mediante prueba (*test*).

Cuando se realiza una evaluación mediante modelado, los métodos que se utilizan difieren significativamente en función de si el sistema se considera con la fiabilidad estable o creciente. Se puede definir de la forma siguiente:

- Fiabilidad estable: cuando se mantiene la capacidad del sistema para entregar el servicio correcto (identidad estocástica de los tiempos sucesivos hasta la avería).
- Fiabilidad creciente: cuando se mejora la aptitud del sistema de entregar el servicio correcto (crecimiento estocástico de los tiempos sucesivos hasta la avería).

La evaluación de la Confiabilidad en sistemas con fiabilidad estable se compone usualmente de dos fases. La primera de ellas es la construcción del modelo del sistema, a partir de procesos estocásticos elementales que modelan el comportamiento y las interacciones de los componentes del sistema. La segunda es el procesamiento del modelo para la obtención de las expresiones y valores de las medidas de la Confiabilidad del sistema.

Los modelos de fiabilidad creciente, sean relativos al *hardware*, al *software*, o al conjunto de los dos, están destinados a la realización de predicciones de la fiabilidad a partir de datos relativos a averías pasadas del sistema.

La evaluación puede realizarse respecto a los fallos físicos, los de diseño, o una combinación de ambos. La Confiabilidad de un sistema es altamente dependiente de su entorno, tanto del físico como de su carga.

Cuando se evalúa un sistema tolerante a fallos, la efectividad de los mecanismos de procesamiento de los errores y de tratamiento de los fallos tiene una influencia primordial; su evaluación puede realizarse mediante modelado o mediante prueba, llamado en este caso **inyección de fallos**.

---

[4] Del inglés, *Performability*.

2.1.4.4 Dependencias entre los medios para alcanzar la Confiabilidad

En las definiciones de Prevención, Tolerancia, Eliminación y Predicción de fallos realizadas previamente, se utiliza la palabra "cómo" para especificar los objetivos que cada mecanismo pretende alcanzar. En la realidad, debido a la imperfección de la naturaleza humana, la cual interviene en todas las actividades realizadas en aquellos mecanismos, todos estos objetivos no son completamente alcanzables. Esto hace que la aplicación de uno de estos medios implique habitualmente la necesidad de aplicar uno o más de los restantes, ya que pueden haberse introducido efectos colaterales en el sistema.

Debido a este motivo, es necesaria la utilización combinada de varios de estos métodos para lograr un sistema de funcionamiento garantizado. Las dependencias existentes entre los diferentes medios se pueden especificar así:

- A pesar de la Prevención, en los diseños se generan fallos, por lo que es necesaria la Eliminación de fallos: cuando se detecta un error durante la verificación, es necesario un diagnóstico para determinar sus causas y eliminarlas.
- La Eliminación de fallos es imperfecta, como lo son los componentes del sistema (*hardware* y *software*). Por este motivo es necesaria la Predicción de fallos.
- La importancia de los sistemas informáticos en la vida cotidiana conduce a establecer unos requisitos de Tolerancia a fallos basados en reglas constructivas. De nuevo, debido a la intervención humana, son necesarias la Eliminación y la Prevención de fallos.

Hay que señalar que el proceso real es aún más recurrente, puesto que debido a la elevada complejidad de los sistemas actuales, son necesarias herramientas para su diseño y construcción. Para que el trabajo realizado con estas herramientas sea el esperado, éstas tienen que ser de funcionamiento garantizado, y así sucesivamente.

Los anteriores razonamientos ilustran la fuerte relación existente entre la Eliminación y la Predicción de fallos. Por este motivo, ambas se incluyen en el término general Validación. La validación de un sistema puede ser de dos maneras:

- Teórica, cuando se realiza una Predicción de fallos sobre un modelo analítico del sistema.
- Experimental, cuando se lleva a cabo una Predicción de fallos sobre un modelo de simulación o un prototipo del sistema, o cuando se realiza una Eliminación de fallos (aplicada igualmente sobre un modelo experimental o un prototipo).

Estos aspectos serán tratados con mayor detalle en el apartado 2.1.8.

2.1.5 Confiabilidad y Tolerancia a fallos

En lo que concierne al desarrollo de sistemas con funcionamiento garantizado, el estado del arte consiste en efectuar una elección sistemática y equilibrada entre diferentes técnicas de Tolerancia a fallos, con el fin de reforzar los métodos de Prevención de fallos [Arlat90a].

La Prevención de fallos enfatiza el uso de componentes fiables, y pretende asegurar que el sistema desarrollado esté exento de fallos. A nivel de *hardware*, son importantes la introducción de protecciones contra las perturbaciones del entorno y la utilización de componentes de alta escala de integración [Siewiorek82]. En lo que respecta al *software*, los métodos principales se concretan en una concepción estructurada y modular, y en el empleo de lenguajes de alto nivel [Courtois92].

Debido a las limitaciones en la formalización y el control de la complejidad tecnológica actual, la Tolerancia a fallos mantiene un papel preponderante en la búsqueda de un nivel significativo de Confiabilidad. La introducción de la Tolerancia a fallos en el proceso de desarrollo de un sistema informático hace necesario:

- La determinación de los tipos de fallos susceptibles de activarse en la fase operativa.
- El diseño de un sistema que utilice mecanismos de redundancia para reducir los efectos de dichos fallos sobre el servicio proporcionado en la fase operativa.

2.1.6 Confiabilidad y Validación

La Validación constituye uno de los principales problemas asociados al desarrollo y explotación de sistemas informáticos con funcionamiento garantizado. Además del aspecto funcional, debe ponerse el acento sobre la confianza en el comportamiento apropiado de los mecanismos que contribuyen a la Confiabilidad, es decir, sobre la **Validación de la Cobertura**. Esto corresponde a una recursión del tipo Validación de la Validación: *¿Cómo tener confianza en los métodos y mecanismos empleados para conseguir la confianza en el sistema?* [Laprie85].

La *Validación de la Cobertura* concierne principalmente a la validación del sistema desarrollado, y por tanto de los mecanismos de tolerancia a fallos integrados para asegurar la Confiabilidad. Dos tipos de parámetros fundamentales permiten cuantificar la eficacia de estos mecanismos: el Factor de Cobertura y la Latencia de Tratamiento. A título de ejemplo, para los mecanismos de detección, se pueden definir de la siguiente manera:

- El **Factor de Cobertura de Detección** es la probabilidad condicional de detección de errores.
- La **Latencia de Detección de error** es el intervalo de tiempo que separa la activación de un fallo en forma de error y su detección.

Es interesante resaltar la importancia de extender los métodos de validación a las diferentes fases de desarrollo (especificación, diseño e implementación), así como en fase operativa. Estos métodos pueden agruparse en dos grandes clases: la Eliminación de fallos y la Predicción de fallos. La Eliminación de fallos consiste en reducir (mediante Verificación) la presencia de fallos e identificar las acciones más apropiadas para mejorar la concepción del sistema. La Predicción de fallos tiene por objetivo principal estimar (mediante Evaluación) la influencia de la aparición, presencia y consecuencias de los fallos sobre el funcionamiento y la Confiabilidad del sistema en fase operacional. La separación entre la Verificación y la Evaluación es en realidad menos marcada de lo que en principio puede parecer, y su complementariedad es un hecho en numerosas ocasiones.

2.1.7 Tolerancia a fallos y Validación experimental

En los apartados anteriores se han enumerado diferentes aspectos de los sistemas tolerantes a fallos reales, como la obtención de los Coeficientes (o Factores) de Cobertura y los Tiempos de Latencia en la detección y en la recuperación de errores, que indican que su Validación precisa de una parte experimental. Este hecho viene motivado por la complejidad en el comportamiento de los sistemas informáticos tolerantes a fallos, debida principalmente a dos motivos [Arlat90a, Arlat92b, PGil92]:

- La especialización y novedad de los componentes y las aplicaciones informáticas, tanto en el *hardware* como en el *software*.

  En cuanto al *hardware*, debido al creciente avance de la tecnología, la utilización de componentes de una determinada "generación" tiene una validez temporal reducida, que hace que las experiencias obtenidas en cuanto a los tipos de fallos y a la patología de los mismos, sirvan de poco en diseños posteriores. Otros aspectos que hay que considerar son la cada vez mayor complejidad de los componentes así como la creación de componentes para aplicaciones empotradas.

  En el *software* se observa una situación parecida, no solamente en cuanto a los lenguajes de programación usados, sino también a las metodologías de programación.

Por otra parte, los sistemas tolerantes a fallos suelen ser utilizados en aplicaciones específicas, con un pequeño número de unidades construidas, lo que dificulta aún más el disponer de datos experimentales acerca de su comportamiento.

- Las incertidumbres relativas a la patología de los fallos, ya que a causa de la complejidad de los sistemas informáticos, existen muchos interrogantes respecto al comportamiento de un sistema en presencia de fallos, sobre todo en el aspecto de cómo cuantificar la influencia de éstos en la Confiabilidad.

Como consecuencia de dichos factores, los modelos de sistemas tolerantes a fallos han evolucionado desde los llamados macroscópicos [Bouricius69], en alusión a un nivel de detalle que sólo tiene en cuenta los procesos de ocurrencia de fallos y reparaciones en los componentes del sistema, hasta los que consideran el comportamiento de los sistemas de tratamiento de fallos, que se denominan microscópicos [Dugan89]. Los modelos macroscópicos se pueden resolver utilizando datos estadísticos de los fabricantes de los circuitos del sistema, teniendo el gran inconveniente de la inexactitud de sus resultados, dado el tratamiento demasiado superficial del proceso de ocurrencia de los fallos. Además, su aplicación para el cálculo de la Confiabilidad del *software* del sistema es muy compleja [Kanoun89].

Por el contrario, los modelos microscópicos están basados en la utilización de procesos estocásticos, e introducen técnicas de resolución analítica y/o de simulación. Se pueden aplicar al conjunto *hardware*/*software* del sistema tolerante a fallos, consiguiendo resultados más exactos de la Confiabilidad. Al tener en cuenta el comportamiento de los sistemas de tratamiento de errores, precisan de datos experimentales para calcular los Coeficientes de Cobertura y los Tiempos de Latencia en la detección y recuperación de los errores, parámetros de una importancia fundamental en el cálculo de la Confiabilidad de los sistemas tolerantes a fallos. Esto explica la necesidad de los métodos experimentales, tanto para el cálculo de la Confiabilidad (Predicción de fallos) como para un mejor conocimiento de la patología de los fallos, que permitirá optimizar los mecanismos de tolerancia a fallos introducidos en el sistema informático (Eliminación de fallos).

2.1.8 Validación experimental e Inyección de fallos

La validación experimental puede llevarse a cabo de dos formas diferentes [Arlat90a]:

- Mediante experiencias no controladas, observando el comportamiento en fase operativa de un sistema informático en presencia de fallos. De este modo se pueden recoger datos sobre Coeficientes de Cobertura, número de averías y coste temporal de las operaciones de mantenimiento.
- Mediante experiencias controladas, analizando el comportamiento del sistema en presencia de fallos introducidos deliberadamente.

El primer método tiene la ventaja de que es más real, pues los fallos que puedan ocurrir en el sistema son observados durante el funcionamiento normal del mismo. Sin embargo, presenta una serie de inconvenientes que lo hacen impracticable en la mayoría de los casos:

- La bajísima probabilidad de ocurrencia de los sucesos bajo observación, sobre todo si se trata de un sistema tolerante a fallos. Esto hace que el número de fallos sea muy bajo para un tiempo aceptable, o que el tiempo de observación requerido sea demasiado largo si se desea realizar una estadística con un margen de confianza adecuado.
- El número reducido de sistemas tolerantes a fallos existentes, ya que son sistemas para aplicaciones especiales. Esto hace todavía más difíciles las observaciones en experimentos no controlados.
- La disparidad de las soluciones adoptadas para aumentar la Confiabilidad en los sistemas tolerantes a fallos, lo que complica la clasificación de estos sistemas para su estudio en presencia de fallos.

Estos inconvenientes hacen que el segundo método, denominado **inyección de fallos**, sea más adecuado para la validación de sistemas tolerantes a fallos. Esta técnica se define como [Arlat90a] *la técnica de validación de la Confiabilidad de Sistemas Tolerantes a Fallos consistente en la realización de experimentos controlados donde la observación del comportamiento del sistema ante los fallos es inducida explícitamente por la introducción (inyección) voluntaria de fallos en el sistema.* La inyección de fallos posibilita la validación de los sistemas tolerantes a fallos en los siguientes aspectos:

- En el estudio del comportamiento del sistema en presencia de fallos, permitiendo:
  - ⇨ Confirmar la estructura y calibrar los parámetros (cobertura, tiempos de latencia) de los modelos microscópicos del sistema.
  - ⇨ Desarrollar, en vistas de los resultados, otros modelos microscópicos más acordes con el comportamiento real del sistema.
- En la validación parcial de los mecanismos de tolerancia a fallos introducidos en el sistema. Se pueden llegar a resultados del tipo: "el X% de los errores del tipo Y son detectados y/o recuperados para una carga del tipo Z".

La inyección de fallos se puede realizar tanto en sistemas *hardware* como *software*. En el *hardware*, los fallos se pueden inyectar en simulaciones del sistema, o en su implementación física, tanto externamente como internamente. En el *software*, los fallos se pueden inyectar en simulaciones del sistema *software*, como por ejemplo simulaciones de sistemas distribuidos, o en sistemas *software* en ejecución, desde los registros de la CPU hasta el nivel de red.

La inyección de fallos constituye un complemento indispensable de otros métodos de validación existentes (prueba formal, prueba simbólica, evaluación analítica, etc.). Existe complementariedad en vez de competencia [DBENCH01, FIT02d], ya que para conseguir un máximo de confianza en el proceso de validación, es necesario aplicar conjuntamente varios métodos, puesto que la utilización aislada de un método no es suficiente para asegurar un buen nivel de Confiabilidad. En relación con esto, cabe citar las metodologías de concepción integrando validación formal, modelado analítico e inyección de fallos utilizadas en los proyectos *SIFT* [Schwartz83], *EVE* [Arlat84], *DELTA_4* [Powell88], *IPDS* (1 y 2) [Randell95], *HIDE* [Majzik98], *GUARDS* [Powell99, Powell01] y *DBench* [DBENCH03]. En la actualidad se está observando una tendencia a utilizar las técnicas de validación formal en la validación de las especificaciones (como ocurre en el proyecto *FAST* [FAST01]), mientras que la validación de la Confiabilidad del sistema se realiza exclusivamente mediante técnicas de inyección (este es el caso del proyecto *FIT* [FIT02c]).

En la Figura 5 [PGil92] se puede apreciar el proceso de validación de un sistema tolerante a fallos mediante inyección de fallos, tanto desde el punto de vista teórico como experimental.

La figura se divide en dos organigramas, (I) y (II), correspondientes respectivamente a la realización de una Predicción de fallos y de una Eliminación de fallos. A su vez, estos dos organigramas se pueden recorrer por dos tipos de caminos, etiquetados como (1) y (2), y que se corresponden respectivamente con operaciones teóricas o experimentales.

Para efectuar una validación teórica hay que seguir el camino (1) del organigrama (I). A partir de unas especificaciones del sistema en desarrollo se construye en primer lugar un modelo teórico, clásicamente de *Markov* o de Redes de *Petri*. A continuación hay que caracterizar el modelo, disponiendo las coberturas y los tiempos de latencia. Estos datos se obtienen, al nivel teórico, bien por hipótesis o por comparación con otros modelos. Éste es uno de los puntos más importantes en el estudio de un sistema: determinar la cobertura y latencia de los errores.

Una vez caracterizado el modelo, se procede a su resolución. Los datos aquí obtenidos se pueden utilizar posteriormente para posibles modificaciones del sistema tolerante a fallos, mediante la realimentación marcada con (c) en la Figura 5.

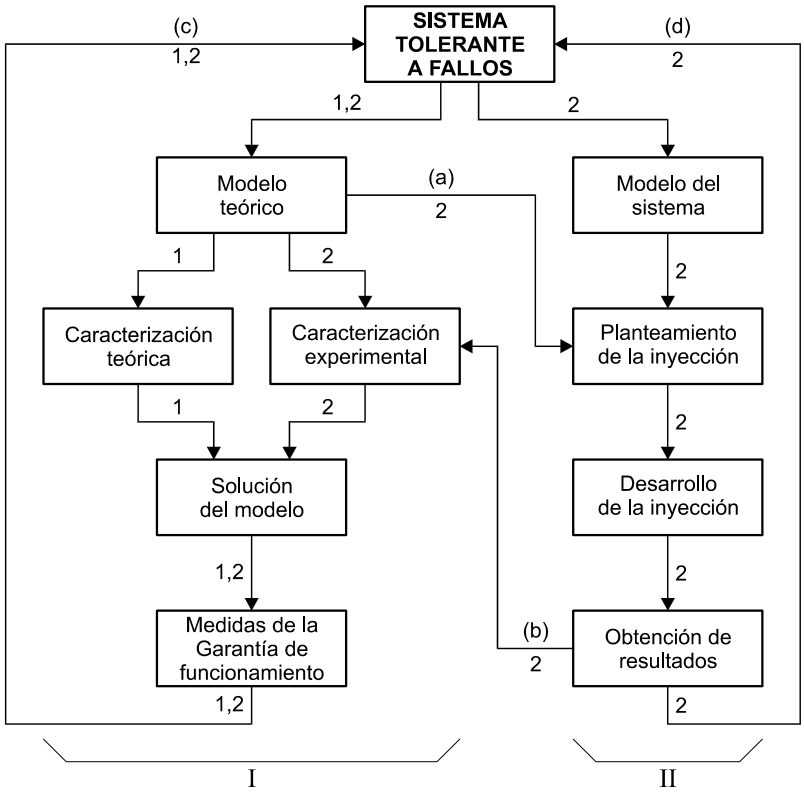


Figura 5. Diagrama de bloques del proceso de validación teórica y experimental mediante inyección de fallos. I: Predicción de fallos. II: Eliminación de fallos.

Los pasos que hay que seguir para llevar a cabo una validación experimental dependen de si la Validación se realiza mediante Predicción o Eliminación de fallos.

Para realizar la validación mediante Predicción de fallos hay que utilizar los organigramas (I) y (II). En este caso, se debe seguir el camino (2) del organigrama (I) para la realización del modelo teórico, si bien éste puede diferir del realizado para una Predicción de fallos teórica. Otra diferencia entre ambas maneras de hacer la Predicción de fallos es la caracterización del modelo, ya que ahora depende del organigrama (II): hay que realizar un modelo experimental del sistema (que puede ser un prototipo o un modelo de simulación), y a partir de las especificaciones del modelo teórico (flecha (a)) se plantea la inyección. Con los resultados obtenidos (coberturas y tiempos de latencia) se finaliza la caracterización experimental del modelo teórico (flecha (b)), tras lo cual se puede resolver y calcular las medidas de la Confiabilidad del sistema, que al igual que en el caso teórico, se pueden utilizar para corregir el sistema (realimentación (c)).

Si la validación se efectúa mediante Eliminación de fallos, sólo hay que seguir el organigrama (II). En este caso, el planteamiento de la inyección sólo depende del modelo experimental del sistema. Con los valores de las coberturas y tiempos de latencia obtenidos, se pueden tomar las medidas oportunas sobre el sistema a través de la realimentación (d).

2.2 Técnicas de inyección de fallos

2.2.1 Introducción

Atendiendo a diferentes puntos de vista, las técnicas de inyección de fallos se pueden clasificar según la clase de sistema sobre la que se aplica (sistema real/modelo), la clase de fallos que se inyectan (físicos o en *hardware*/*software*), o la clase de mecanismo que realiza la inyección (*hardware*/*software*). Todas estas clasificaciones se pueden agrupar en una, que divide las técnicas de inyección en tres grandes grupos [Arlat92a, Jenn94a, Clark95, Iyer95, Hsueh97, Yu03]:

- **Inyección física**. Mediante esta técnica se pueden inyectar fallos físicos en el sistema real (o en un prototipo) utilizando un mecanismo físico. El proceso de inyección se realiza actuando sobre la estructura material del sistema, alterando su funcionamiento a través de medios físicos. Se le denomina también **HWIFI**, del inglés *Hardware Implemented Fault Injection*.
- **Inyección implementada por *software***. Esta técnica se aplica también sobre el sistema real o un prototipo, pero utiliza mecanismos lógicos (programas, el sistema operativo, etc.) para realizar la inyección. En este caso, es posible inyectar tanto fallos físicos como *software*. Se pueden distinguir dos subtécnicas:
  - ⇨ **Técnicas de mutación**. Esta técnica de inyección permite inyectar fallos que representan fallos reales producidos en el *software*. Se realiza en el *software* para validar la prueba de *software* [DeMillo87].
  - ⇨ **Emulación de fallos en el *hardware* mediante el *software*** (o SWIFI, del inglés *Software Implemented Fault Injection*). Básicamente, se trata de modificar la memoria del *software* en ejecución (tanto la zona de programa como la de datos) y los registros del procesador accesibles por *software* [Iyer95, Campelo99].
- **Inyección mediante simulación**. Actúan sobre modelos del sistema en diferentes niveles de abstracción [DGil99a]. Dependiendo del nivel de representación del modelo se pueden inyectar tanto fallos físicos como *software*. Tradicionalmente, el medio utilizado para realizar la inyección ha sido una aplicación que simula el modelo. Sin embargo, en la actualidad está teniendo cierto auge la emulación del modelo en FPGA.

Esta clasificación puede verse resumida en la Figura 6. Hay que tener en cuenta, sin embargo, el instante en el cual se pueden aplicar las distintas técnicas de inyección dentro del ciclo de vida de un sistema (especificación, diseño, prototipado y operación)[5]. Así, mientras en la fase de diseño sólo se pueden aplicar técnicas de simulación, en la fase de prototipado sólo es posible utilizar las técnicas de inyección sobre el sistema real o un prototipo del mismo, es decir, las técnicas HWIFI y/o SWIFI [Iyer95].

La inyección de fallos durante la fase de diseño comprueba la efectividad de los mecanismos de tolerancia a fallos y evalúa la Confiabilidad del sistema, permitiendo a los diseñadores aplicar un proceso de realimentación temporal. La simulación requiere, no obstante, precisión en los parámetros y en la validación de los resultados. Aunque la estimación de los parámetros se puede realizar a partir de medidas reales del pasado del sistema, los cambios en el diseño y la tecnología dificultan la obtención y/o el uso de estas medidas.

La inyección de fallos simulada se puede efectuar a diferentes niveles de abstracción: eléctrico, lógico, sistema, etc. Sus objetivos son determinar los "cuellos de botella" en la Confiabilidad, las coberturas de los mecanismos de detección/recuperación, la efectividad de los

[5] Existen otras nomenclaturas [Calvez93] en las que se consideran al menos cinco fases: especificación, diseño, implementación, producción y operación.

esquemas de reconfiguración, la pérdida de prestaciones y otras medidas. La realimentación que introduce la simulación puede ser muy útil para disminuir el coste del rediseño del sistema.

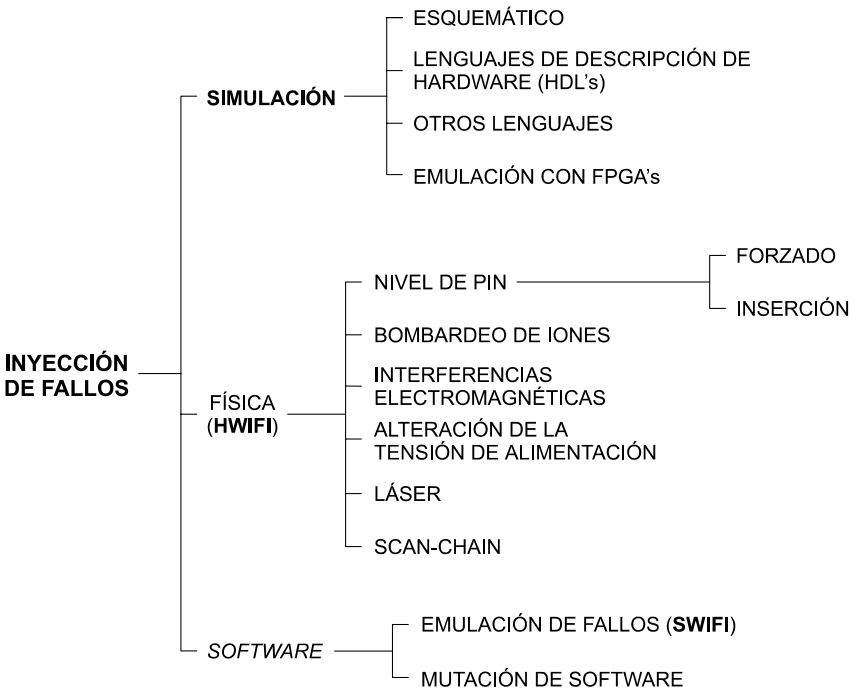


Figura 6. Clasificación general de las técnicas de inyección de fallos.

Durante la fase de prototipado, el sistema funciona bajo condiciones de carga controladas. Para evaluar en esta etapa el comportamiento del sistema ante fallos, incluyendo la cobertura de detección y la capacidad de recuperación de varios mecanismos de tolerancia a fallos, se emplean HWIFI y SWIFI. La inyección de fallos en el sistema real puede proporcionar información sobre el proceso de los fallos, desde su aparición hasta la recuperación del sistema, incluyendo las coberturas y las latencias de propagación, detección y recuperación. Pero este tipo de inyección de fallos sólo puede estudiar fallos artificiales. No puede suministrar ciertas medidas importantes de la Confiabilidad, como el tiempo medio entre averías (en inglés, *Mean Time Between Failures*, MTBF) y la Disponibilidad (en inglés, *Availability*).

En esta fase, aunque los objetivos de la inyección son similares a los de la inyección mediante simulación, los métodos difieren radicalmente debido a que la inyección y la monitorización son reales. Los fallos se pueden inyectar a nivel de *hardware* (fallos lógicos) o de *software* (corrupción del código o de los datos). También se pueden utilizar técnicas de radiación de iones pesados para inyectar fallos y estresar el sistema.

En los apartados siguientes se describirán las características generales de las diferentes técnicas de inyección, así como algunos trabajos publicados y herramientas construidas que implementan automáticamente dichas técnicas de inyección durante el proceso de Validación.

2.2.2 Inyección de fallos implementada mediante hardware

Esta técnica, denominada también HWIFI (del inglés *Hardware Implemented Fault Injection*) introduce fallos en el sistema computador con la ayuda de instrumentación adicional. Esta técnica es adecuada para estudiar características de la Confiabilidad que requieren una resolución temporal elevada, como la latencia de los fallos en la CPU, y que no se pueden obtener con otros métodos de inyección de fallos. En la Tabla 1 se muestran las principales ventajas e inconvenientes de esta técnica.

Ventajas	Inconvenientes
▪ Trabaja muy bien cuando se necesita gran precisión temporal, tanto en el disparo de la inyección como en la monitorización del sistema. ▪ La evaluación experimental mediante la inyección en el *hardware* es, en muchos casos, la única forma de calcular con precisión las coberturas y latencias del sistema. ▪ Esta técnica es la que mejor se adapta a los modelos de fallos en bajo nivel. ▪ Los experimentos de inyección son muy rápidos, y además, se pueden ejecutar casi en tiempo real, permitiendo la posibilidad de ejecutar un gran número de experimentos de inyección. ▪ La ejecución de experimentos de inyección de fallos en el *hardware* real, el cual está ejecutando el *software* real, permite incluir cualquier fallo de diseño que pueda estar presente en el diseño actual del *hardware* y del *software*. ▪ Los experimentos de inyección de fallos se realizan utilizando el mismo *software* que se ejecutará en la aplicación real. ▪ No es necesario desarrollar y validar un modelo del sistema.	▪ Existe un gran riesgo de dañar el sistema bajo prueba. ▪ Los altos niveles de integración, los sistemas SOC (del inglés *Systems On a Chip*) y las tecnologías con gran densidad de integración limitan la accesibilidad de la inyección. ▪ Algunos métodos *hardware* de inyección de fallos necesitan parar el procesador para inyectar el fallo y volver a ponerlo en marcha, lo que no es efectivo a la hora de medir latencias en sistemas físicos. ▪ Presenta una baja portabilidad entre diferentes sistemas. ▪ El conjunto de puntos de inyección así como de fallos inyectables está limitado. ▪ El tiempo de configuración para cada experimento puede contrarrestar el tiempo ganado al realizar los experimentos en tiempo real o cercanos al tiempo real. ▪ Se necesita *hardware* específico para realizar los experimentos de inyección de fallos. ▪ Observabilidad y controlabilidad limitadas.

Tabla 1. Ventajas e inconvenientes de la inyección de fallos implementada mediante *hardware* [Yu03].

Todos sabemos que un fallo físico es un fenómeno físico adverso, tanto interno (cortocircuitos, etc.) como externo (perturbaciones medioambientales, temperatura, vibraciones, etc.) que *ataca* al sistema bajo estudio. Si se toman como referencia los límites de un circuito integrado, se pueden distinguir dos tipos de técnicas de inyección de fallos: inyección externa e inyección interna.

2.2.2.1 Inyección de fallos externa

Una de las técnicas HWIFI más frecuente es la inyección de fallos a nivel de *pin*. Su objetivo es alterar los valores lógicos de distintas señales sobre las que se efectúa la inyección [Schmid82, Schuette86, Finelli87, Damm88, Arlat90a, PGil92, Madeira94, PGil97, Martínez99, PGil03].

Dentro de la idea de alterar el nivel de los *pines* se destacan a su vez dos modalidades: forzado, en la que el fallo a inyectar se fuerza directamente en el *pin*, sin extraer ningún componente del sistema, e inserción, en la que el circuito se extrae del sistema y se sustituye por otro que inyecta el fallo.

En [Damm86, Gunneflo90, Miremadi92] la inyección se realiza mediante conmutaciones en la fuente de alimentación, en vez de alterar los niveles de uno o más *pines*. Con esta técnica se consigue provocar fluctuaciones de tensión en la entrada de alimentación, siendo posible variar tanto la duración de las perturbaciones como el salto de tensión producido. Otra forma de inyectar fallos en el exterior de un circuito integrado es mediante interferencias electromagnéticas (también conocida como EMI) [Reisinger94, Karlsson95].

A continuación se presentan algunas de las herramientas más destacadas en cuanto a inyección a nivel de *pin* y mediante interferencias electromagnéticas.

2.2.2.1.1 Inyección de fallos a nivel de pin

Tal y como se ha comentado anteriormente, la inyección en los *pines* de los circuitos integrados es una de las técnicas HWIFI más empleadas en la actualidad. Ésta ha servido para validar distintos sistemas tolerantes a fallos, así como para obtener la cobertura de distintos mecanismos de detección de fallos. Con esta técnica los fallos se inyectan directamente en los *pines* de los circuitos integrados que componen el sistema. Por otra parte, la duración de los fallos se puede determinar para que se obtengan fallos transitorios, intermitentes o permanentes, pudiéndose introducir distintos tipos de fallos:

- Pegado a (del inglés *stuck-at*) '0' o a '1'. Los *pines* se fuerzan a uno de esos dos niveles.
- Puenteado (del inglés *bridging*), en la que distintos *pines* se interconectan entre sí.
- Inversión de señales (del inglés *inverted signal*), cuando se invierte el valor de una señal.
- Conexión abierta (del inglés *open connection*), que consiste en dejar en alta impedancia el *pin* sobre el que se inyecta.

Como se indicaba anteriormente, y desde el punto de vista de la implementación, existen dos técnicas de inyección de fallos a nivel de *pin*: forzado (los fallos se inyectan directamente sobre los *pines* del circuito integrado) e inserción (los circuitos integrados se sacan del sistema, aislándolos de esta forma del resto de los componentes, y en su lugar se coloca una pinza especial para inyectar los valores deseados).

En estas dos técnicas sólo se pueden inyectar fallos en los *pines* de los circuitos integrados, siendo impracticable la inyección por debajo del nivel de *pin*. Como normalmente el sistema es una tarjeta del computador, se suele inyectar en los *buses* de la tarjeta y en otras señales accesibles de la misma.

Se han construido distintas herramientas de inyección de fallos que emplean esta técnica, como por ejemplo **FTMP** [Lala83], **MESSALINE** [Arlat90b], **RIFLE** [Madeira94] y **AFIT** [PGil92, PGil97, Martínez99, PGil03] (desarrollada por el Grupo de Sistemas Tolerantes a Fallos, GSTF, de la Universidad Politécnica de Valencia).

Existe también una herramienta comercial denominada **DVT-100** [Proteus96, Stewart97], comercializada por *Proteus Corporation*. Se trata de una herramienta automatizada dotada de sondas (de inyección y monitorización) robotizadas.

2.2.2.1.2 Inyección de fallos mediante interferencias electromagnéticas

Las interferencias electromagnéticas (EMI) existen en muchos ambientes. En entornos de automoción y plantas industriales, por ejemplo, estas interferencias causan distintos fallos en los sistemas informáticos. Por este motivo, una de las técnicas para estudiar el comportamiento ante fallos de diversos sistemas consiste en la generación de estas interferencias de forma controlada. Concretamente, los experimentos EMI se suelen basar en la generación de ráfagas de

aproximadamente 15 ms de duración, con un periodo de 300 ms, de una frecuencia entre 1.25 kHz y 10 kHz y unos niveles de tensión entre los 225 V y los 4400 V. Estas condiciones intentan modelar los ambientes en los cuales conmutan cargas inductivas con relés o contactos mecánicos.

La forma en la que se aplican estas interferencias suele ser, o bien aplicarlas sobre dos placas conductoras entre las que se encuentra el sistema analizado, o directamente sobre el circuito integrado que se quiere analizar, con una punta de prueba diseñada a tal efecto.

En [Karlsson95] se compara esta técnica con otras dos técnicas HWIFI: radiación con iones pesados e inyección a nivel de *pin*. El objetivo es validar el sistema distribuido de tiempo real tolerante a fallos MARS [Reisinger94].

2.2.2.2 Inyección de fallos interna

Los métodos más usados en este tipo de inyección están basados en la radiación de iones, aunque también se han realizado algunos experimentos de inyección mediante láser, y otros que aprovechan los puertos de prueba (TAP, del inglés *Test Access Port*) de los circuitos integrados para acceder a los registros internos (denominados *Scan-Chain*). A continuación se muestran algunos ejemplos.

2.2.2.2.1 Inyección de iones pesados

Este método se basa en causar interferencias en el interior de un circuito integrado utilizando radiación de iones pesados [Gunneflo89, Karlsson89, Gaisler97, Gaisler02, Sivencrona03]. Una ventaja de esta técnica es poder producir fallos transitorios en puntos aleatorios internos del circuito integrado, normalmente fallos de tipo *bit-flip* (inversión del bit) individuales o múltiples. Sin embargo, uno de sus inconvenientes es la baja reproducibilidad de los fallos [Miremadi92, Karlsson94, Karlsson95, Miremadi95].

Para usar esta técnica, al circuito integrado se le debe retirar su encapsulado, y tanto el circuito a inyectar como la fuente de material radioactivo deben ser aislados mediante un entorno al vacío. Esto es necesario debido a que los iones pesados son atenuados por las moléculas de aire y otros materiales.

La radiación habitualmente está generada por una fuente de Cf–252, aunque también es posible utilizar un ciclotrón (o acelerador de partículas). Este segundo método permite controlar la energía de las partículas (regulando la velocidad de impacto) o la localización de los impactos. Además, posibilita la inyección de otros tipos de partículas, como neutrones de diferentes energías. La aplicación de la inyección de fallos con iones pesados presenta serias dificultades técnicas. Además, los iones pesados desencadenan fenómenos que pueden dañar el circuito (*latch-up* y excesiva disipación de calor, entre otros).

Un ejemplo de herramienta es **FIST** (*Fault Injection System for Study of Transient Fault Effects*), que utiliza una fuente de Cf–252. Empleando FIST, en [Gunneflo89, Karlsson89] se investigó la cobertura de los errores y las latencias de detección del microprocesador de 8 bits MC6809. En [Gaisler97, Gaisler02] se muestran los resultados de validar sendos procesadores tolerantes a fallos (ERC32 y LEON-FT) inyectando los fallos mediante un acelerador de partículas. En [Sivencrona03] se utiliza una fuente de Cf–252 para inyectar fallos en un nodo de un sistema distribuido basado en la arquitectura *Time-Triggered* [Kopetz98a, Kopetz98b], validando con estos experimentos el diseño del protocolo *TTP/C* [Kopetz94].

2.2.2.2.2 Inyección mediante láser

En los primeros trabajos efectuados sobre este tipo de inyección de fallos, se generaron fallos permanentes mediante la ruptura de ciertas conexiones internas de los circuitos integrados mediante un rayo láser [Velazco90, Martinet92].

Más recientemente se ha realizado algún estudio inyectando fallos transitorios. En [Samson97, Samson98] se presenta una herramienta de inyección basada en rayo láser. El mecanismo físico involucrado en la aparición de los fallos transitorios es la generación de portadores (pares e^-–h^+) en el substrato de silicio, debido a la energía de la radiación láser. Estos portadores pueden ser atraídos por áreas de difusión de los transistores, y provocar cambios momentáneos en los niveles lógicos. Se trata, por tanto, de un mecanismo similar a los SEU (del inglés *Single Event Upset*), causados por las partículas de alta energía.

El efecto de estos eventos en la operación del circuito depende de la localización en el circuito del dispositivo electrónico afectado, así como de la temporización (instante y duración) del evento respecto del reloj del sistema. Por ejemplo, el cambio momentáneo en la entrada o la salida de un transistor en un biestable puede inducir un cambio en el estado del sistema.

Por lo que respecta al proceso de inyección, primero es necesario determinar la localización precisa de las áreas/difusiones del circuito que se pretenden afectar. Esto se hace utilizando una herramienta CAD de *layout*. A continuación, la localización se traslada a coordenadas X-Y de la tabla del láser. Por último, se dispara el láser, produciendo un pulso corto de suficiente potencia para inducir un fallo transitorio, pero sin dañar el componente. En [Samson98] se presentan algunos experimentos de prueba sobre un procesador RISC con mecanismos de detección internos.

La técnica de inyección mediante láser tiene, respecto a la de iones pesados, la ventaja de una mayor controlabilidad espacial y reproducibilidad de los experimentos. Por el contrario, el proceso de inyección y el instrumental asociado son más complejos.

2.2.2.2.3 Inyección de fallos mediante Scan-Chain

Dentro de la inyección de fallos interna, últimamente se ha recurrido a los mecanismos de prueba de los microprocesadores para alterar el valor de registros internos de los mismos. Así, en la Universidad de Chalmers se ha desarrollado la herramienta **FIMBUL** (*Fault Injection and Monitoring using BUilt in Logic*) [Folkesson98]. A través del TAP (del inglés *Test Access Port*), es capaz de modificar el valor de los bits de los componentes internos del procesador para emular distintos tipos de fallos. En este trabajo se comenta que se podían inyectar fallos transitorios, y se esperaba poder inyectar fallos permanentes. La forma de inyectar está basada en el establecimiento de un punto de parada cuando se desea realizar una inyección y efectuar, cuando esto ocurra, la alteración de alguna unidad funcional del microprocesador. También muestra una comparación entre los resultados alcanzados con esta herramienta y MEFISTO-C (sobre un modelo en VHDL del mismo procesador). Se podría introducir la definición de un nuevo tipo de inyección, denominado "inyección de fallos mediante *Scan-Chain*" (*Scan-Chain Implemented Fault Injection*), o SCIFI.

2.2.3 Inyección de fallos implementada mediante software

El desarrollo tecnológico ha propiciado una creciente densidad de integración de los circuitos integrados, provocando que la comprobación y evaluación de sistemas tolerantes a fallos se haya convertido en algo cada vez más complicado. Muchas de las técnicas físicas descritas en el punto anterior solamente pueden forzar un fallo en los límites de un circuito integrado, es decir, a nivel de *pin*.

Sin embargo, cada vez es más frecuente la inclusión en el mismo encapsulado, sobretodo en entornos basados en microcontroladores, de diversas unidades funcionales (lo que se denomina SOC, del inglés *Systems On a Chip*). Es en este punto, por tanto, donde la inyección de fallos implementada mediante *software* (en inglés *Software Implemented Fault Injection* o SWIFI) cobra cada vez más importancia.

Mientras la inyección implementada mediante *hardware* requiere una circuitería específica de instrumentación y una interfaz con el sistema, la inyección de fallos implementada mediante *software* proporciona una metodología barata y fácil de controlar. Con este método de inyección no se necesita ninguna instrumentación adicional. El usuario puede elegir los puntos de inyección tanto en el *hardware* como en el *software* del sistema, siempre que sean accesibles por el juego de instrucciones de la máquina. Además, esta técnica permite también la emulación de fallos *software* mediante un apropiado cambio en el código.

Se han propuesto diversos métodos para emular diferentes tipos de fallos en el *hardware* y en el *software* [Pradhan96]. Básicamente se trata de modificar la memoria (tanto la zona de programa como la de datos) y los registros accesibles de la CPU.

Cuando se utiliza la inyección SWIFI para emular fallos, usualmente se asume que éstos son transitorios. Por ejemplo, los bits afectados en la memoria o en los registros de la CPU pueden ser sobrescritos por instrucciones posteriores. No obstante, también puede emular fallos permanentes, inyectando repetidamente el mismo fallo en un punto del sistema cada vez que haya un acceso a dicho punto. Por ejemplo, para emular un fallo permanente de tipo *stuck-at* '0' en un bit particular de una palabra de memoria, el bit se fuerza a '0' tras cada operación de escritura en dicha palabra. Para emular un fallo permanente de tipo *stuck-at* '1' en una línea del *bus* de direcciones, el bit correspondiente en la dirección efectiva (en el contador de programa o en un registro de la CPU) se fuerza a '1' antes de cada acceso al *bus*. Evidentemente, esta emulación es costosa en tiempo, introduciendo la monitorización y ejecución de muchas instrucciones adicionales.

La inyección mediante SWIFI puede afectar a aplicaciones del usuario, al sistema operativo, o a ambos, a diferencia de las técnicas de inyección implementadas mediante *hardware*, en las que es difícil de orientar la inyección hacia áreas específicas del *software*. Dependiendo del objetivo de la inyección, el inyector se debe insertar en diferentes sitios. En el caso de una aplicación del usuario, el inyector se inserta en la aplicación o puede ser una capa adicional entre la aplicación y el sistema operativo. El inyector se insertará en el sistema operativo[6] si es aquí donde se van a inyectar los fallos, ya que es bastante complejo añadir una capa adicional entre la máquina y el sistema operativo. Sin embargo, a pesar de la flexibilidad de esta técnica, presenta algunas restricciones:

1. Solamente se pueden inyectar fallos en posiciones accesibles por *software*.
2. El entorno de inyección se debe diseñar cuidadosamente, con el fin de mitigar los efectos de las perturbaciones de la ejecución de la carga del sistema o los cambios que se pueden introducir en la estructura del programa original al insertar las rutinas de inyección.
3. Esta técnica presenta una resolución temporal bastante pobre. Para fallos de latencia grande, como los que se producen en la memoria, la baja resolución temporal puede no ser un problema. En cambio, para los de latencia pequeña, como los producidos en los *buses* y en la CPU, puede haber problemas en la determinación del comportamiento de los errores (por ejemplo en la propagación). Una posible solución a este problema es la utilización de un monitor *hardware*, es decir, empleando una aproximación híbrida [Young92], la cual combina la versatilidad de la inyección implementada mediante *software* y la precisión de la monitorización *hardware*, resultando adecuada para medir latencias muy bajas. No obstante, la monitorización *hardware* diminuye la observabilidad, ya que está limitada a puntos concretos, aumentando además el coste.
4. Todos los métodos de inyección SWIFI están basados en características (*hardware* y *software*) del sistema al que se aplica, por lo que la generalización de una herramienta SWIFI (independientemente del método empleado) a diferentes sistemas es bastante complicada.

[6] Generalmente se realiza en la interfaz que ofrece al usuario, conocida como API (del inglés *Application Program Interface*).

[Folkesson99] presenta una clasificación de los métodos que se siguen para realizar la inyección mediante SWIFI:

- Previo a la ejecución, del inglés *pre-runtime fault injection*. Este tipo de técnica tiene la ventaja de minimizar la intrusión en el sistema. Consiste en modificar la carga del mismo antes de su ejecución. A su vez, existen dos variantes:
  - ⇨ Modificación en tiempo de compilación, que consiste en modificar el código fuente del programa que se desea alterar, introduciendo fallos en el sistema para que se comporte de manera diferente. Esta técnica tiene el inconveniente de requerir el código fuente de la carga que ejecutará el sistema bajo estudio.
  - ⇨ Modificación del código ejecutable. En este caso, se modifican las zonas de datos y/o código de la carga del sistema, antes de su ejecución.
- En tiempo de ejecución, del inglés *run-time falut injection*. En este caso, la inyección se realiza por parte de otro(s) proceso(s) (programa(s)) que se ejecuta(n) en el sistema, independientemente de la carga "real". Estos procesos tienen como misión modificar el entorno del programa, es decir, las zonas de programa y de datos de la memoria y los registros del procesador (evidentemente, aquellos registros accesibles por *software*).

A pesar de que esta técnica se ha aplicado en bastantes sistemas, los resultados publicados han sido realizados ex-profeso para sistemas particulares, debido a que la aplicación de los posibles métodos de inyección SWIFI a cualquier diseño de manera independiente es bastante compleja. A continuación se comentan las herramientas más significativas, así como otros resultados interesantes:

- **FIAT** (*Fault Injection based Automated Testing environment*) [Segall88], desarrollada en la Universidad de Carnegie Mellon, se basaba en una serie de IBM RT PC conectados mediante una red *token-ring*. El objetivo de esta herramienta era la inyección de patrones de error que representaran a los generados tanto en los programas como en los componentes. Los fallos se generaban corrompiendo la imagen de memoria de una tarea, permitiendo al usuario escoger la localización deseada del fallo. En los experimentos realizados se observó que la cobertura media para diferentes cargas estaba en un rango entre el 50 y el 60%. Los inconvenientes de esta herramienta eran la inyección únicamente de fallos transitorios y la imprecisión en la obtención de los tiempos de latencia.
- **EFI**, desarrollado en la Universidad de Dortmund [Echtle92], surgió con la idea de comprobar algoritmos tolerantes a fallos en sistemas distribuidos. Cada nodo del sistema distribuido tenía su propio inyector, localizado entre el nivel de enlace de datos y el nivel que implementaba la tolerancia a fallos, con lo que se podían alterar tanto los mensajes entrantes como salientes. En el mismo centro se desarrolló **ProFI** (*Processor Fault Injector*) [Lovric93], con el objetivo de detectar los errores en el *hardware*. Se basa en inyectar fallos permanentes a nivel de registros y código máquina del procesador.
- El entorno **FERRARI** (*Fault and ERRor Automatic Real-time Injector*) [Kanawati92, Kanawati95] fue desarrollado en la Universidad de Texas con el objetivo de evaluar sistemas complejos emulando fallos físicos. Se implementó sobre una estación SUN SPARC, realizándose la inyección de fallos mediante la corrupción del estado de ejecución de un programa, de tal manera que el estado sería el mismo que si hubiera sido causado por un error interno. Esto se consigue gracias a la utilización de interrupciones internas que se pueden activar bajo demanda del usuario. De esta forma, FERRARI puede emular un gran número de fallos físicos, así como errores de control de flujo, permitiendo inyectar tanto fallos permanentes como transitorios.
- **SFI** (*Software Fault Injector*) [Rosenberg93], de la Universidad de Michigan, se aplicaba a sistemas distribuidos. Esta herramienta soportaba inyección a bajo nivel en nodos y a alto nivel entre los nodos del sistema, permitiendo inyectar fallos transitorios, intermitentes y permanentes, pudiendo además especificar los parámetros temporales y

los distintos tipos de fallos. Podía inyectar fallos tanto en el procesador como en la memoria y el sistema de comunicaciones, cambiando para ello algunas de las instrucciones generadas por el compilador (inyección *pre-runtime*). A partir de SFI se desarrolló **DOCTOR** (*integrateD sOftware implemented fault injeCTiOn enviRonment*) [Han94], con el objetivo de aumentar su portabilidad, minimizando su dependencia del sistema (tanto del *hardware* como del sistema operativo).

- **FINE** (*Fault Injection and moNitoring Environment*) [Kao93] se desarrolló en la Universidad de Illinois con el objetivo de estudiar la propagación de los fallos en el núcleo del sistema operativo UNIX. Para ello, FINE era capaz de emular fallos permanentes, transitorios e intermitentes en el procesador, la memoria y los *buses*, e inyectar fallos de programa en el núcleo del sistema.

  Con el objetivo de facilitar la inyección y analizar la propagación del error, el inyector y la monitorización de los fallos se encuentran empotrados en el núcleo del sistema. La inyección de fallos se apoya en las interrupciones del sistema, pudiéndose considerar al inyector de fallos como una capa más del sistema actuando entre el sistema operativo y la máquina. Algunos resultados de los experimentos realizados indican que los fallos en la memoria y en los programas tienen una latencia grande, mientras que en el *bus* y la CPU es pequeña. Cerca del 90% de los errores fueron detectados por los mecanismos *hardware*. De éstos, cerca de la mitad se detectaron en los datos, cuando el sistema intentaba acceder a una zona de memoria para la que no tenía permiso.

  DEFINE [Kao94] es una evolución de FINE que incluye capacidades distribuidas. Modifica los ficheros ejecutables para emular fallos de memoria, insertando interrupciones *software* en el segmento de código, siendo capaz también de inyectar fallos en la CPU y en el *bus*.

- **FTAPE** (*Fault Tolerance And Performance Evaluator*) [Tsai95a, Tsai95b], se ha desarrollado en la Universidad de Illinois con el objetivo principal de poder comparar sistemas tolerantes a fallos. Entre sus características principales destacan el uso de una carga sintética para generar actividad, principalmente en la CPU, la memoria o el sistema de entrada/salida y la elección de dónde y cuándo inyectar. Al ser la carga configurable, se crean en la máquina lo que denominan "condiciones de estrés", surgiendo el concepto de inyección de fallos basada en el estrés del sistema (del inglés *stress-based injection*), que garantiza una mayor efectividad del fallo en función del sistema o de su uso.

- **Xception** [Carreira98] ha sido desarrollada en la Universidad de Coimbra. Esta herramienta es capaz de inyectar fallos permanentes y transitorios sobre el procesador, la memoria y los *buses* del sistema, sin tener que modificar el fichero ejecutable, ya que gracias a las capacidades internas de depuración de los procesadores es posible lanzar una rutina de interrupción tras un determinado tiempo o evento. Esta herramienta utiliza los avances en depuración y monitorización que existen en algunos procesadores modernos para inyectar fallos y analizar los resultados. La sobrecarga es mínima, aunque lógicamente esta herramienta sólo puede utilizarse con procesadores que ofrezcan estas capacidades de depuración.

- **MAFALDA** (*Microkernel Assessment by Fault injection AnaLysis and Design Aid*) [Rodríguez98, Fabre99] se desarrolló en el LAAS-CNRS de Toulouse con el objetivo de validar sistemas basados en *Microkernels* COTS. MAFALDA es capaz de corromper los parámetros de invocación a las llamadas del sistema y de inyectar fallos en el propio núcleo. La corrupción de los parámetros de las llamadas al sistema da a conocer la robustez del mismo, mientras que las inyecciones (tanto en el espacio de código como en el de datos) ayudan a estudiar la propagación del error a través de los componentes internos.

  La corrupción de las llamadas al sistema se realiza mediante la alteración de uno de los bits de los parámetros de forma aleatoria, mientras que la inyección de fallos se basa en rutinas de interrupción que, en función de la inyección deseada, alteran aleatoriamente

algunas posiciones de memoria. Esta herramienta es capaz de realizar tanto inyecciones permanentes como transitorias.

- **EXFI** (*EXception-based Fault Injector*) [Benso98b, Benso98c] utiliza las excepciones del procesador para realizar la inyección. La aplicación de EXFI se realiza sobre una placa comercial M68KDIP de *Motorola*, con un microprocesador M68040. La principal ventaja de esta técnica es su bajo coste, dado que no se requiere ningún accesorio (ni *hardware* ni *software*). Sin embargo, tiene como gran inconveniente su intrusividad y un relativamente alto factor de retardo.

  A partir de EXFI se ha desarrollado la herramienta **FlexFi** [Benso99b], que permite inyectar fallos mediante SWIFI utilizando tres métodos de inyección diferentes:

  ⇨ Utilizando excepciones [Benso98b, Benso98c].

  ⇨ Híbrido [Benso98a]. Los fallos se inyectan a través de interrupciones provocadas por una placa externa. De esta manera se reduce la intrusividad, a costa de tener que desarrollar circuitería.

  ⇨ Basado en el modo de diagnóstico [Benso99a]. Los nuevos microprocesadores y microcontroladores producidos por *Motorola* disponen de un modo de funcionamiento especial, denominado modo de diagnóstico (en inglés *Background Diagnostic Mode* o BDM).

- **SOFI** (*SOftware Fault Injector*) [Campelo99] es un inyector de fallos por *software* desarrollado por el Grupo de Sistemas Tolerantes a Fallos de la Universidad Politécnica de Valencia. Esta herramienta ejecuta la inyección durante la ejecución de la aplicación del sistema bajo análisis (técnica *run-time fault injection*), utilizando un pequeño agente de inyección que se tiene que montar junto con la aplicación del sistema bajo prueba. Este agente se encarga de recibir las órdenes de un supervisor, que de forma aleatoria en el tiempo determina el instante de la inyección.

  SOFI intenta emular fallos en la ALU, los registros del procesador y la memoria mediante la alteración de un bit en el segmento de código (es decir, de una instrucción) o en el segmento de datos. Es, por tanto, el nivel más bajo en el que se puede realizar la inyección. En cuanto a la persistencia del fallo, SOFI es capaz de inyectar fallos permanentes y transitorios. La duración de los fallos transitorios se puede configurar por el usuario entre un ciclo máquina (100 ns) hasta aproximadamente 13 ms, siendo el tipo de fallo inyectado el *bit-flip*.

  Una de las características más interesantes y novedosas de SOFI es su alta efectividad. Cuando SOFI inyecta en el segmento de código consigue una efectividad del 100%. Esto quiere decir que todo fallo inyectado va a ejercitar los mecanismos de detección de errores implementados en el sistema bajo estudio. Por último, otro dato destacable es su alta velocidad de inyección, consiguiendo aproximadamente 1000 fallos inyectados por hora.

- **UMLinux** (*User Mode Linux*) [Sieh02, Höxer02] se presenta como una herramienta de inyección de fallos en sistemas bajo Linux conectados en red. La herramienta utiliza la función `ptrace`, de manera similar a la herramienta mostrada en [Sieh93]. En [Höxer02] se muestra un ejemplo de aplicación de UMLinux sobre una red local virtual.

- **INERTE** (*Integrated NExus-based Real-Time fault injection tool for Embedded systems*) [Yuste03] es una herramienta desarrollada por el Grupo de Sistemas Tolerantes a Fallos de la Universidad Politécnica de Valencia durante el proyecto de investigación europeo *DBench* [DBENCH03] (IST-2000-25425), y aplicada en la inyección de fallos en sistemas empotrados de tiempo real. Está basada en *Nexus* [Nexus99], una interfaz estándar de depuración que permite inyectar fallos en tiempo real sin ninguna intrusión. Con INERTE se pueden inyectar fallos transitorios (la inyección de fallos permanentes está bajo estudio) en la memoria del sistema sin introducir ninguna sobrecarga temporal.

La monitorización del sistema tras la inyección de los fallos se realiza mediante un dispositivo *hardware*, generalmente un emulador del microprocesador o microcontrolador bajo estudio.

- Otros herramientas y aportaciones:
  - ⇨ **CSFI** (*Communication Software Fault Injector*) [Carreira95] se implementó durante el proyecto europeo FTMPS Nº 6731 con el fin de inyectar fallos de comunicaciones en computadores paralelos. En este trabajo se muestran los resultados de su aplicación a un *transputer* T805 bajo PARIX 1.2. CSFI es capaz de corromper los mensajes generados en la comunicación entre los procesadores del *transputer*. La inyección puede afectar a cualquier sección de cualquier tipo de mensaje.
  - ⇨ **ORCHESTRA** [Dawson96a, Dawson96b] se ha desarrollado en la Universidad de Michigan para su aplicación a sistemas distribuidos en tiempo real. Se implementa como un inyector de fallos a nivel de protocolo, instalándolo entre dos capas del protocolo de comunicaciones. Esta nueva capa de inyección de fallos (en inglés *Protocol Fault Injector* –PFI– *layer*) intercepta y manipula los mensajes que pasan a su través.
  - ⇨ En [Fuchs96] se presenta otro ejemplo de inyección de fallos mediante SWIFI (antes de la ejecución) aplicado al sistema de tiempo real MARS (*MAintainable Real-Time System*). En este trabajo se muestran los resultados de tres estudios realizados: la efectividad de los mecanismos de detección de errores, la necesidad de los mecanismos de detección de errores en el nivel de aplicación y la existencia de violaciones de la premisa de silencio en caso de avería (del inglés *fail-silence*).
  - ⇨ En [Stott97, Stott98] se aplica SWIFI para el análisis de la Confiabilidad de *Myrinet*, una red comercial de alta velocidad. La inyección se lleva a cabo mediante una aplicación que modifica las instrucciones que se ejecutan en la parte de comunicaciones, escribiendo en la memoria de la interfaz. En estos trabajos se muestran los resultados de dicha aplicación, y en [Stott98], además, se comparan con los obtenidos al aplicar inyección de fallos mediante simulación sobre un modelo del mismo sistema. En este caso, la herramienta de inyección utilizada es DEPEND (véase el apartado 2.2.4.1.5). La comparación da un 83% de concordancia en los comportamientos producidos en ambos casos. La elevada discrepancia se justifica por la simplicidad del modelo realizado para la inyección mediante simulación.
  - ⇨ **THESIC** (*Testbed for Harsh Environment Studies on Integrated Circuits*) [Velazco00a, Velazco00b] se aplica a sistemas empotrados. Esta herramienta se ha implementado sobre un soporte físico compuesto por una placa base para las operaciones de control y que es la interfaz con el usuario, y una placa secundaria para adaptar el sistema sobre el que se va a inyectar al protocolo de la placa base. La comunicación entre ambas placas se lleva a cabo mediante una memoria, siendo necesario construir una placa secundaria por cada sistema que se analice con THESIC.

La siguiente tabla (Tabla 2) muestra las principales ventajas e inconvenientes de la inyección de fallos basada en *software*.

Ventajas	Inconvenientes
▪ Esta técnica puede ser aplicada tanto a aplicaciones como a sistemas operativos. ▪ Los experimentos de inyección se pueden ejecutar casi en tiempo real, permitiendo la posibilidad de ejecutar un gran número. ▪ La ejecución de los experimentos de inyección en el *hardware* real, el cual está ejecutando el *software* real, tiene la ventaja de incluir cualquier fallo de diseño que pueda estar presente en el diseño actual del *hardware* y del *software*. ▪ No se requiere ningún *hardware* específico. ▪ El coste de implementación es bajo. ▪ No es necesario desarrollar y validar un modelo del sistema.	▪ Conjunto limitado de instantes de inyección. ▪ No es posible inyectar fallos en localizaciones inaccesibles al *software*. ▪ Requiere la modificación del código fuente para soportar la inyección de fallos, lo que significa que el código que se ejecuta durante los experimentos de inyección no es el mismo que se ejecutará durante el funcionamiento real del sistema. ▪ Observabilidad y controlabilidad limitadas, ya que sólo se puede acceder a aquellas partes del sistema que sean accesibles por *software*. ▪ Es muy difícil modelar fallos permanentes. ▪ La ejecución del *software* inyector de fallos podría afectar a la planificación de las tareas del sistema de tal manera que podría causar pérdidas de *deadlines*.

Tabla 2. Ventajas e inconvenientes de la inyección de fallos implementada mediante *software* [Yu03].

2.2.4 Inyección de fallos mediante simulación

En la inyección de fallos mediante simulación un modelo del sistema bajo prueba, el cual puede estar desarrollado en diferentes niveles de abstracción, es simulado en otro sistema. La inyección se realiza mediante la alteración de los valores lógicos de los elementos del modelo durante la simulación.

Las técnicas actuales de desarrollo de sistemas digitales utilizan de forma extensiva la simulación durante las primeras fases de diseño del mismo, siendo posible utilizar la simulación para efectuar un primer análisis de las Prestaciones y la Confiabilidad. Este primer análisis permite ahorrar tiempo y dinero en el caso de detectar errores en el sistema[7]. En comparación con modelos analíticos, la simulación presenta la capacidad de modelar sistemas complejos con un alto grado de fiabilidad, sin tener que realizar ningún tipo de restricción para conseguir que el modelo analítico sea matemáticamente tratable. Otra ventaja de la inyección de fallos mediante simulación respecto de otras técnicas de inyección es la gran observabilidad y controlabilidad de todos los componentes modelados.

En la Figura 6 se muestran varios tipos de técnicas de inyección de fallos basadas en simulación. Estas técnicas, a su vez, se pueden dividir en dos grupos. En primer lugar, la simulación de esquemas eléctricos y la de modelos en lenguajes de algún tipo (HDL, C, C++, ADA, etc.) son técnicas basadas en una simulación por *software* de un modelo, mientras que la emulación de fallos con FPGA (también conocida como *Emulación de fallos* y *Emulación lógica*) utiliza como instrumento de simulación del sistema un soporte físico, circuitos lógicos programables (PLD, del inglés *Programmable Logic Devices*) del tipo FPGA (del inglés *Field Programmable Gate Array*).

[7] En EDA (*Electronic Design Automation*) es célebre la "regla de los diez" (en inglés *rule-of-tens*), que determina que el coste de encontrar y corregir defectos se multiplica por 10 en cada etapa del proceso [Kilty95].

2.2.4.1 Inyección de fallos basada en simulación software

En el primer grupo de técnicas, la inyección de fallos que se realiza para el análisis de la Confiabilidad depende de los diferentes niveles de abstracción definidos. Por ejemplo, en [Pradhan96] se consideran tres niveles:

- Eléctrico. Se corresponde con los circuitos eléctricos, inyectándose fallos que consisten en alteraciones de corriente y/o voltaje.
- Lógico. Se corresponde con las puertas lógicas, y se inyectan fallos de los tipos *stuck-at* (‘0’ y ‘1’) y *bit-flip*.
- Funcional. Correspondiente con bloques funcionales, los fallos inyectados representan cambios en los registros de la CPU, *bit-flip* en la memoria, etc.

Sin embargo, estos niveles varían según diferentes autores. En [Jenn94a] se especifican otros niveles: tecnológico, de circuito, lógico, de transferencia de registros (RT, del inglés *Register Transfer*) y no interpretado, mientras que [Siewiorek94] considera también tres niveles: de circuito, lógico y de sistema. Es decir, el establecimiento de los niveles de abstracción presenta bastante diversidad, y en ocasiones se trata de una cuestión de terminología, estando a menudo los diferentes niveles de abstracción solapados en las diferentes clasificaciones [DGil99a].

La complejidad de las simulaciones (tanto en espacio como en tiempo) depende de los niveles de abstracción utilizados. Cuanto mayor sea el nivel, se tendrá menor complejidad en las simulaciones a costa de perder detalles en la simulación. Por ejemplo, la simulación al nivel eléctrico no se puede usar de manera efectiva para estudiar sistemas VLSI complejos, y la simulación al nivel lógico no permite el estudio de sistemas computadores grandes. La simulación a nivel de sistema posibilita el análisis de las características de los computadores grandes y las redes. La desventaja obvia de aumentar el nivel de abstracción es que los modelos (tanto del sistema bajo estudio como de los fallos que se inyectan) se alejan de los mecanismos físicos reales.

Otro factor importante a tener en cuenta son las características temporales de los fallos. En [Iyer86, Siewiorek94] se demuestra que más del 80% de las averías de los sistemas computadores son debidas a fallos transitorios. Estos fallos tienen diferentes causas físicas, como transitorios en la alimentación, diafonía[8] capacitiva o inductiva, o radiación cósmica.

Existen algunos problemas comunes que afectan a la inyección de fallos en todos los niveles de abstracción [Pradhan96]. En primer lugar se debe evitar la “explosión” del tiempo de simulación. Esta explosión puede ocurrir cuando se simulan muchos detalles o cuando se necesita una simulación larga para obtener resultados estadísticos significativos ya que la probabilidad de los fallos es extremadamente pequeña. Respecto a los fallos, hay que tener en cuenta cuál es el modelo de fallos apropiado para el nivel de abstracción elegido. Para un modelo y un tipo de fallo dado, se debe estudiar cuidadosamente el lugar de la inyección del mismo, ya que puede suceder que el fallo afecte a partes no sensibilizadas por la carga de trabajo o que los fallos inyectados en un módulo determinado presenten un impacto similar. Por último, el impacto de los fallos en la Confiabilidad del sistema también depende de los programas de prueba o carga de trabajo (del inglés *workloads*). Por esta razón, el análisis del sistema debe realizarse mientras éste ejecuta cargas representativas, que pueden ser aplicaciones reales, *benchmarks* selectivos o programas sintéticos.

La Tabla 3 muestra las principales ventajas e inconvenientes de la inyección de fallos basada en simulación *software*.

[8] En inglés, *crosstalk*.

Ventajas	Inconvenientes
▪ Puede soportar todos los niveles de abstracción. ▪ No se presenta ningún tipo de intrusión en el sistema real. ▪ Control total de los modelos de fallos y de los mecanismos de inyección. ▪ Los cambios en el sistema son muy sencillos. ▪ Coste bajo en la automatización. ▪ No requiere ningún *hardware* específico. ▪ Proporciona a los diseñadores del sistema una realimentación a tiempo. ▪ Máxima controlabilidad y observabilidad. ▪ Validación del sistema durante la fase de diseño. ▪ Capaz de inyectar fallos transitorios, permanentes e intermitentes. ▪ El coste temporal necesario para la inyección del fallo es nulo.	▪ Es necesario un gran esfuerzo para desarrollar el modelo. ▪ Gran consumo temporal durante las simulaciones. ▪ Los modelos no están siempre disponibles. ▪ Los resultados dependen de la precisión del modelo. ▪ Los modelos pueden no incluir todos los fallos de diseño que pueden estar presentes en el *hardware* real. ▪ Los resultados dependen tanto de la bondad del modelo del sistema como de la representatividad de los modelos de fallos.

Tabla 3. Ventajas e inconvenientes de la inyección de fallos basada en simulación *software* [Yu03].

A continuación se presenta un análisis de diferentes estudios y herramientas de inyección de fallos mediante simulación. En primer lugar se indican las que utilizan un soporte *software* para simular el modelo del sistema, agrupadas por diferentes niveles de abstracción. A continuación, se describen las diferentes aproximaciones realizadas siguiendo la técnica de emulación de fallos con FPGA. En este capítulo no se trata la inyección de fallos mediante simulación sobre modelos VHDL, a pesar de que el trabajo principal de esta tesis se ha realizado con esta técnica de inyección. Se ha preferido tratarla más detalladamente en el siguiente capítulo: "Técnicas de inyección de fallos basadas en VHDL".

2.2.4.1.1 Nivel tecnológico

Los trabajos de [Choi93] presentan un modelado de los procesos de electromigración y de ruptura de la capa de óxido en los circuitos VLSI. La integración del proceso tecnológico de electromigración se realiza en dos fases principales:

1. Se simula el circuito, almacenando las informaciones concernientes a las conmutaciones que han tenido lugar.
2. Las informaciones obtenidas son utilizadas por el modelo de electromigración para evaluar las degradaciones de las conexiones y, en definitiva, la existencia de conexiones averiadas.

Para modelar la degradación de la capa de óxido se emplea un proceso similar.

2.2.4.1.2 Nivel de transistor

Dependiendo del autor, este nivel se denomina de diferentes formas: circuito, eléctrico o de dispositivo [Jenn94a, Siewiorek94]. La inyección de fallos a este nivel se puede usar para estudiar el impacto de las causas físicas que producen los fallos y los errores. A pesar de que los modelos de fallos más establecidos son el *stuck-at* y el *bit-flip*, utilizando este nivel algunos estudios muestran que estos modelos de fallos no son siempre representativos de los fallos físicos [Galiay80, Shen85, DBENCH02, Gracia02a]. Además, en algunos circuitos es necesario

el uso de simuladores de fallos que manejen fallos eléctricos transitorios y fallos físicos permanentes si estos contienen elementos analógicos y digitales que no puedan ser caracterizados completamente por los modelos de fallos de tipo lógico.

Sin embargo, bajar a este nivel de detalle implica una serie de restricciones desde el punto de vista de la simulación, en la que difícilmente se puede estudiar el comportamiento de más de un circuito integrado. Además, el tiempo de simulación necesario restringe el número de componentes que se pueden simular.

FOCUS [Choi92] modela el impacto de la penetración de una partícula ionizada mediante la inserción de una fuente de corriente a nivel de circuito. El modelo, escrito en el lenguaje SPLICE3, permite el estudio de la propagación de los errores originados a través de diferentes niveles de abstracción: eléctrico, lógico y algorítmico.

2.2.4.1.3 Nivel lógico

También se denomina nivel de puerta (del inglés *gate-level*) [Siewiorek94]. Los fallos y las funciones de los circuitos son simulados mediante modelos lógicos abstractos. La simulación en este nivel realiza operaciones binarias que representan el comportamiento de un dispositivo dado, obteniéndose salidas con valores discretos, así como también información temporal aproximada. Habitualmente se comparan las salidas del sistema con y sin fallos para determinar la ocurrencia de los fallos y de los errores.

Los primeros trabajos se basan en la inyección de fallos *stuck-at* de tipo permanente. Así, en [Lomelino86] este tipo de inyección se aplica sobre una descripción esquemática basada en puertas y biestables del procesador *bit-slice* AMD 2901, utilizando un simulador desarrollado por la NASA. En [Czeck90] se lleva a cabo sobre las señales de un modelo del procesador IBM RT PC realizado en Verilog, otro lenguaje de descripción de *hardware* (HDL) cuyo uso también está muy extendido.

En [Cha93, Cha96] se describe una herramienta que hace referencia a una serie de elementos que permiten una simulación eficaz de fallos al nivel lógico, representativos de los fallos transitorios materiales (impacto de iones). En una primera fase se obtiene un modelo lógico de los fallos mediante simulación a nivel de circuito para poder utilizar, en una segunda fase, un simulador de fallos temporal. En el momento en que los errores son almacenados en un registro del sistema, se efectúa una tercera y última fase de simulación de fallos paralela con "retardo–cero".

En [Choi93] se presenta otro método para generar modelos de fallos reales al nivel lógico, denominado aproximación del diccionario de fallos (del inglés *fault-behavior dictionary approach*). En primer lugar se simula al nivel eléctrico la zona donde se inyectan los fallos (por ejemplo con PSPICE). Mientras se efectúa la inyección, sobre el subcircuito (formado por los puntos que rodean el lugar de inyección) se aplican todas las combinaciones de entrada, analizando el comportamiento de las salidas del subcircuito, y almacenándolo en un diccionario. La entrada del diccionario consiste en el vector de entradas y el lugar, instante y duración de la inyección. Después se realiza la simulación al nivel lógico a partir del modelo de fallos del diccionario. La idea es atractiva porque podría extenderse para generar fallos en otros niveles de mayor abstracción.

2.2.4.1.4 Nivel de transferencia entre registros (nivel RT)

En este nivel, la simulación se encamina hacia la validación de los distintos mecanismos *hardware* que incorporan los procesadores con respecto a la alteración de alguno de los registros internos. Por ejemplo, en [Ohlsson92, Rimén92b, Rimén93b] se analiza el comportamiento ante fallos de un modelo en VHDL de un procesador de 32 bits con *watchdog* interno. El modelo de fallo adoptado en este estudio es el *bit-flip*, mientras que en [Karlsson90] se realiza una comparación de la técnica de inyección de iones pesados con la inyección simulada.

En [Vargas99a, Vargas00a, Vargas00b] se añaden diferentes técnicas de codificación al modelo de circuito bajo prueba para aumentar la fiabilidad del mismo, mediante una herramienta CAD que denominan **FT-PRO**. Esta herramienta genera elementos de memoria tolerantes a fallos transitorios, en especial tolerantes a SEU (del inglés *Single Event Upset*). Se centran en este tipo de fallo debido a que son los más comunes en aplicaciones espaciales y de aviónica, e incluso, serán muy comunes a nivel del mar debido al uso de tecnología submicrónica. Una vez generado el circuito con esta herramienta, inyectan fallos modificando el código, creando una especie de mutante que inyecta el fallo (en este caso *bit-flips*) según un MTBF[9] determinado previamente.

2.2.4.1.5 Nivel de sistema

La simulación de fallos a este nivel se utiliza para estudiar sistemas computadores completos así como redes de computadores, en lugar de sus componentes individuales. Estos estudios habitualmente consideran como un todo el *hardware*, el *software*, sus interacciones y la interdependencia entre los diversos componentes del sistema. Sin embargo, existen algunos problemas en el desarrollo de modelos de simulación a este nivel, como puede ser el esfuerzo y el tiempo requerido para desarrollar un modelo de simulación, la dificultad para establecer los modelos de fallos adecuados, la existencia de un amplio espectro de componentes, el impacto del *software* en la Confiabilidad y la "explosión" del tiempo de simulación.

Las herramientas de simulación a nivel de sistema se diferencian de las herramientas de modelado analítico en la distinta aportación que hacen al análisis de la Confiabilidad. Las herramientas de modelado analítico sólo utilizan modelos probabilísticos para representar el funcionamiento del sistema, es decir, básicamente caracterizan el efecto de un fallo en el sistema mediante un conjunto de probabilidades y distribuciones. Por el contrario, la simulación a nivel de sistema únicamente necesita conocer la tasa de llegada y los tipos de fallos, no precisando la caracterización del efecto de los fallos. Por lo tanto, los resultados de la simulación pueden identificar los mecanismos de las averías, obtener su probabilidad y cuantificar el efecto de los fallos, lo que permite la extracción de las características que se desea modelar, y ayudan a determinar y especificar la estructura y los parámetros de los modelos analíticos. A continuación se describirán algunas herramientas de simulación a nivel de sistema:

- **NEST** (*NEtwork Simulation Testbed*) es un entorno gráfico bajo UNIX cuyo objetivo es modelar, ejecutar y monitorizar sistemas distribuidos y protocolos [Dupuy90]. El usuario puede desarrollar modelos de simulación y redes de comunicaciones utilizando para ello un conjunto de herramientas gráficas. El modelo incluye las funciones de los nodos y el comportamiento de los enlaces. El usuario puede también borrar o añadir nodos y enlaces (de esta manera se pueden emular las averías), o cambiar sus propiedades mientras se efectúa la simulación, pudiendo utilizar estas simulaciones reconfigurables dinámicamente para estudiar el impacto de las averías en los nodos o enlaces, así como su recuperación.
- **DEPEND** [Goswami92, Goswami97] implementa una aproximación intermedia entre el modelado interpretado y el no interpretado. Esta herramienta propone una biblioteca de clases de objetos C++ elementales (como votadores, procesadores, memorias, inyectores de fallos, canales de comunicación, etc.) o complejos (como servidores, TMR, etc.), a partir de los cuales el usuario construye su modelo por instanciación y composición. La aproximación "orientada a objetos" es interesante porque permite la descripción más detallada de los componentes y el paso progresivo de un modelo no interpretado a un modelo interpretado.
- **REACT** [Clark92] permite la descripción de sistemas compuestos por un conjunto de procesadores y bloques de memoria interconectados, con mecanismos de detección y tratamiento de errores (votador, código detector/corrector de error, etc.). El

[9] MTBF: *Mean Time Between Failures*, o tiempo medio entre averías.

comportamiento de un procesador en presencia de fallos se modela con una tasa de ocurrencia de errores sobre los *buses* de datos y de direcciones. Los errores debidos al *software* son tenidos en cuenta añadiendo la tasa de avería de los programas a la tasa de errores del procesador. A nivel de la memoria y de la lógica de detección y tratamiento de errores, los fallos afectan a la lógica de decodificación de las direcciones y a la zona de almacenamiento.

- En [Aylor92] los métodos propuestos se aplican sobre un modelo no interpretado, basado en un formalismo próximo a las redes de *Petri*, y construido sobre el lenguaje VHDL, permitiendo esta aproximación la integración en un mismo modelo de submodelos interpretados y no interpretados. Aquí, la inyección de un fallo consiste simplemente en declarar la presencia de un fallo en una marca. La herramienta **ADEPT** [Kumar94, Ghosh95c] utiliza este formalismo para efectuar la evaluación de los parámetros de sistemas distribuidos, analíticamente y mediante simulación.
- Otras herramientas y aportaciones:
  - ⇨ En [Smith96] hacen hincapié en la generación de un conjunto mínimo de fallos de tal manera que toda inyección produzca un fallo con el fin de acortar el tiempo de simulación. Este recorte del tiempo de simulación lo consiguen eliminando del conjunto de fallos a inyectar aquellos fallos que no se activan. Otra ventaja de este método es la reducción de las simulaciones requeridas debido al colapso de fallos (del inglés *fault collapsing*), es decir, que un único fallo represente varios fallos.
  - ⇨ El modelo de comportamiento consiste típicamente, a nivel de sistema, en un conjunto de procesos que se ejecutan en un sistema distribuido. Por tanto, un punto importante del sistema reside en el protocolo de comunicaciones, que también ha de ser probado contra fallos. De esta forma, para determinar fallos de diseño del protocolo, algunos estudios combinan el análisis de la evolución del flujo del código y la inyección de fallos en los mensajes [Echtle91, Chen93].
  - ⇨ En [Maxion93] se describe una serie de experimentos para detectar y diagnosticar la capacidad del sistema para tratar fallos en redes de área local. Por otra parte, en [Jagannath95] se estudia el impacto de los fallos en el nivel de enlace de datos de la red.
  - ⇨ Aparte de estudiar las redes de comunicación, también se estudian las propiedades de los nodos, es decir, cómo se comportan los mecanismos y algoritmos de tolerancia a fallos. En [Avresky92] se utiliza la inyección de fallos para analizar y mejorar el comportamiento de distintos algoritmos que tratan los fallos que ocurren en el sistema. En [Goswami93] se inyectan fallos en el espacio de memoria del sistema mientras se simula la ejecución de los programas.
  - ⇨ **POLIS** [Lajolo00] es un entorno de codiseño que integra la inyección de fallos. Distingue entre inyección de fallos comportamental y arquitectural. La primera analiza el comportamiento del sistema en presencia de fallos transitorios, mientras que la segunda se encarga de analizar el comportamiento tanto del *software* como del *hardware* del sistema, también en presencia de fallos transitorios.
  - ⇨ **BOND** [Baldini03] simula el comportamiento anormal de un computador ejecutando *Windows* NT4.0/2000. Esta herramienta es capaz de inyectar fallos en *software* COTS[10] sin modificar ni el sistema operativo ni la aplicación. Además, las inyecciones se realizan en diferentes localizaciones, a cualquier nivel del contexto de la aplicación.

[10] COTS: del inglés *Commercial Off-The-Shelf.*

2.2.4.2 Inyección de fallos basada en emulación

Aunque la emulación de fallos con FPGA es conceptualmente muy diferente de las técnicas recién explicadas, se ha decidido incluirla en este apartado porque para llegar a realizar la inyección de los fallos, se parte de un modelo del sistema descrito en un lenguaje de descripción de *hardware*, siendo esta técnica una especie de evolución de la inyección de fallos basada en este tipo de lenguajes.

Esta técnica también se denomina *Emulación de fallos*[11] [Cheng95] y *Emulación lógica* [Hwang98]. Está basada en el uso de FPGA para realizar prototipos preliminares del sistema en desarrollo, utilizándose habitualmente lenguajes de descripción de *hardware* (VHDL o Verilog, principalmente) para la especificación del modelo que se implementará en las FPGA, así como herramientas de síntesis para poder mapear y programar las FPGA. Con estos prototipos se persigue que puedan representar parcialmente el comportamiento del sistema, sin pretender ni que cumplan toda su funcionalidad ni mucho menos que tengan el comportamiento temporal esperado en el diseño final [Leveugle01a].

Por estas razones, la utilidad principal de esta técnica es la localización temprana de fallos de diseño, que reduzcan en gran medida las combinaciones de prueba en etapas posteriores del diseño, bien mediante inyección de fallos o aplicando vectores de prueba, ganándose velocidad a la hora de realizar los experimentos pertinentes.

En las primeras aproximaciones a esta técnica [Cheng95, Li95, Burgun96, Hong96, Li97, Cheng99], se utilizaba un emulador lógico compuesto por varias placas con FPGA en paralelo, así como el *software* de control necesario para poder programar (configurar) las FPGA y hacerlas funcionar en modo traza (como un simulador *software*).

La primera técnica de inyección de fallos, también denominada inyección de fallos estática, consistía en detener el funcionamiento del sistema en un momento determinado y reconfigurar las FPGA. La reconfiguración podía ser de todas las FPGA, de un conjunto de ellas o solamente de una FPGA. Además, la reconfiguración podía ser total (la FPGA entera) o parcial (sólo algunas partes de la FPGA). El principal inconveniente de esta implementación es la gran cantidad de tiempo perdido en la reconfiguración.

Para evitar las excesivas pérdidas de tiempo del método anterior, surgió la inyección dinámica, consistente en modificar el diseño original para hacerlo "inyectable" (en inglés *fault-injectable*). Con este método, en primer lugar se añaden al modelo entradas adicionales de control que indicarán si se inyecta un fallo o no, y dónde. En segundo lugar, se modifica el diseño para que la FPGA pueda ejecutar tanto la función correcta del mismo como las versiones erróneas. La activación de cada "versión" se regula mediante una entrada de control. De esta manera, con un registro de desplazamiento circular se puede seleccionar de manera sucesiva cada una de las "versiones" de la función, y estudiar el comportamiento del sistema en su conjunto.

Con el aumento de la densidad de integración y la capacidad de procesamiento de las FPGA, han aparecido en el mercado placas de prototipado[12] (o de desarrollo) que se pueden conectar directamente a un computador "maestro" [Leveugle01a]. Además, su bajo coste (comparado con el de un sistema basado en emulador) y la eliminación del *software* emulador han aumentado la capacidad de automatización de esta técnica. Sin embargo, las placas de desarrollo presentan un nuevo tipo de problema: el número de *pines* de entrada/salida disponibles para implementar funciones es bajo, ya que este tipo de tarjetas suelen disponer de dispositivos adicionales (visualizadores, señales de reloj, interruptores, pulsadores, *leds*, etc.) preconectados a la FPGA. Para solventar el problema, en [Leveugle01a] se propone añadir al modelo sendos elementos adicionales para que hagan de interfaz de entrada y salida. A través de ellos se

[11] No confundir con SWIFI (ver apartado 2.2.3).

[12] Este tipo de placas pueden ser comercializadas tanto por las propias empresas fabricantes de FPGA [Lima01] como por otras empresas (terceras partes) [Leveugle01a].

pueden multiplexar los *pines* disponibles para utilizarlos como entradas y salidas. El tamaño (en número de líneas) y conexión de estos elementos adicionales es dependiente del modelo, aunque es posible automatizar su inserción.

En cuanto al modo en que se realiza la inyección de los fallos, se pueden distinguir varios métodos. En primer lugar, en [Leveugle00a, Leveugle01a] se presenta una inyección de fallos basada en la generación de mutantes, mientras que en [Lima01] la inyección se basa en perturbadores, en ambos casos al nivel RT. Estas técnicas de inyección (mutantes y perturbadores) se explicarán con más detalle en los apartados 3.2.2.2 y 3.2.2.1.

La siguiente aproximación está basada en mutantes a nivel de puerta. En este caso, es necesario que el modelo sea estructural a nivel de puerta[13] para tener acceso a los biestables. Posteriormente, los biestables del modelo inicial a nivel de puerta se sustituyen por otros capaces de inyectar fallos a través de unas señales de control. Sin embargo, una vez realizado este proceso, existen diferencias en los modelos, y en consecuencia, en los métodos para ejecutar la inyección de fallos. El primer método [Velazco01a, Velazco01b] presenta una circuitería de inyección puramente combinacional, mientras que en el segundo método [Civera01a, Civera01b] el mutante del biestable es más sofisticado, conteniendo además de la circuitería de inyección, un segundo biestable conectado en serie con los demás biestables, de manera que constituyen una cadena. Esta cadena se puede escribir y leer a través de dos líneas. En el momento de la inyección, en función del contenido de los biestables secundarios se conmuta el contenido de los primarios. Es decir, se implementa un mecanismo similar al *Scan-Chain*, una técnica HWIFI descrita en el apartado 2.2.2.2.3. Este método es mucho más potente que el anterior, ya que permite inyectar fallos múltiples.

En [Civera01b] se presenta **FIFA** (*Fault Injection by means of FPGA*). En este caso, así como en [Civera01a, Civera01d] el modelo se instrumenta, es decir, el modelo VHDL original es modificado para incorporar la instrumentación necesaria para la inyección. A la hora de realizar la inyección, es necesario recargar el modelo en la FPGA para cada conjunto de experimentos, con la consiguiente pérdida de tiempo. Para evitar estas pérdidas de tiempo, existe otra línea de trabajos [Antoni02, deAndrés03, Parreira03] en los que la inyección se realiza mediante una "reconfiguración dinámica" con el fin de evitar tanto la modificación del modelo como el tiempo de recarga del mismo. En este caso se utiliza la capacidad de reconfiguración en tiempo de ejecución que presentan algunas familias de FPGA para modificar dinámicamente y en función del fallo a inyectar, la configuración de la FPGA.

Por último, en [Alderighi03] se presenta una herramienta que permite inyectar fallos en el mecanismo de control de la configuración de FPGA de tipo *Virtex*, a diferencia de los métodos explicados anteriormente, en los que los fallos se inyectan en la configuración de celdas de memoria y en registros de usuario. La inyección se realiza mediante la modificación del *bitstream* de configuración mientras éste se está descargando en la FPGA, consiguiendo que la herramienta de inyección sea independiente de la herramienta de síntesis. Con este tipo de inyección se consigue estudiar el efecto de fallos de tipo SEU que afectan a la correcta configuración del dispositivo. Además, se puede acceder y analizar el efecto de éstos fallos en cualquier registro de configuración de la máquina de estados finitos. La herramienta permite el análisis de fallos simples y múltiples del tipo SEU que afectan al mecanismo de control de la configuración. La inyección se realiza mediante otra FPGA, siendo ésta controlada por un PC. Esta FPGA inyectora modifica el *bitstream* mientras se está cargando en el sistema bajo prueba (que es otra FPGA). La modificación consiste simplemente en el cambio del valor lógico del bit de configuración. Esta modificación implica también recalcular el CRC del *bitstream*.

Para acabar con este punto, la Tabla 4 muestra las ventajas e inconvenientes de la inyección de fallos basada en emulación.

[13] Dicho modelo estructural puede ser el diseño original o haber sido obtenido a partir de la síntesis de un modelo comportamental.

Ventajas	Inconvenientes
▪ Es muy rápida. ▪ Puede realizarse durante las primeras etapas del diseño del sistema (cuando el modelo sintetizable esté disponnible). ▪ Existe la posibilidad de conectar la FPGA al entorno real (en inglés, *in-system emulation*). ▪ Permite mayor accesibilidad, un mayor conjunto de modelos de fallos y un análisis más exacto de la patología de los fallos que la inyección de fallos implementada mediante *hardware*.	▪ Al ser circuitos sintetizables, sólo se puede utilizar un subconjunto del VHDL estándar. ▪ No se puede inyectar en todas las partes del modelo. ▪ Existe un número limitado de emuladores comerciales. ▪ Se necesita modificar el código original, con lo que este código modificado puede no cumplir los requerimientos del código original, tanto temporales (la frecuencia máxima del reloj en la FPGA depende del código introducido, por lo que se debe optimizar el código escrito), como espaciales (el nuevo código tiene que "caber" en la FPGA). Es decir, se debe evitar el *overhead* temporal y espacial. ▪ Lectura de datos restringida por el emulador (número máximo de E/S). ▪ Es necesaria la reconfiguración de la FPGA, y en algunos casos, descargar el fichero de configuración. ▪ Se permiten muy pocos modelos de fallos (básicamente el *bit-flip*).

Tabla 4. Ventajas e inconvenientes de la inyección de fallos basada en emulación [Leveugle00b, Yu03].

2.3 Inyección de fallos híbrida

En los últimos años, con el auge de sistemas más complejos que los "simples" sistemas basados en un computador (sistemas distribuidos, redes de comunicaciones, sistemas en tiempo real, sistemas empotrados, etc.), se están acentuando las carencias de las diferentes técnicas de inyección. Por estas razones, se está dando un giro en la política de los diferentes grupos de investigación, y se empieza a observar una tendencia al desarrollo de herramientas que permiten aplicar diferentes técnicas de inyección sobre el mismo sistema y utilizando una interfaz única. Se puede definir la inyección de fallos híbrida como aquella solución que combina dos o más técnicas de inyección de fallos, con el fin de aprovechar las ventajas de ambas técnicas. A continuación se presentan algunos trabajos.

En [Güthoff95] se presenta una herramienta híbrida denominada *Mixed-Mode Fault Injection* que combina la inyección de fallos basada en *software* (o SWIFI, ver apartado 2.2.3) con la inyección de fallos basada en simulación. El objetivo de esta herramienta es la aceleración del proceso de inyección (de manera similar a ASPHALT, tal y como se verá a continuación), realizándose el modelo en este caso a nivel de sistema. Para realizar la inyección, en primer lugar se realiza una ejecución normal, utilizando el prototipo del sistema. A la hora de inyectar el fallo, se pasa el estado del prototipo al modelo, en el cual se realiza la inyección. No hace falta pasar el valor de todos los registros o celdas de memoria del prototipo al modelo, ya que no se utilizan todos estos valores durante la simulación. Unicamente se pasan aquellos valores que utiliza el simulador. Por último, se pasa el estado del modelo con el fallo inyectado al prototipo, ejecutándose la inyección. Realizan también un análisis previo del sistema y de la carga de trabajo para eliminar aquellas inyecciones que no van a perturbar al sistema, y por tanto, su análisis no será de ninguna utilidad.

ASPHALT [Yount96] utiliza una metodología híbrida para realizar la inyección rápida de fallos transitorios en el *hardware* del sistema. Consiste en la combinación de la técnica SWIFI con la simulación al nivel RT de un modelo en Verilog. El sistema estudiado es el microprocesador ROMP (IBM *Risc-Oriented MicroProcessor*), un microprocesador RISC con estructura *pipeline*. La técnica SWIFI garantiza la rapidez al ejecutarse sobre el sistema real, mientras que la simulación RT incrementa la precisión de los modelos de fallos. Los modelos de fallos utilizados en la simulación son:

- Carga de registro no efectuada (*missed load*).
- Carga de registro efectuada erróneamente (*extraneous load*).
- Modificación del contenido de los registros (*bit-flip*, cargar todo a '0', cargar todo a '1').

La validez de los modelos de fallos ha sido contrastada comparando la simulación RT con una simulación al nivel lógico con fallos *stuck-at* transitorios, utilizando una aproximación similar a la del *diccionario de fallos* comentada anteriormente (ver apartado 2.2.4.1.3). En el estudio se indica que aproximadamente el 97% de los fallos del nivel lógico son cubiertos por los fallos RT.

Ya se comentó en el punto 2.2.3 la herramienta EXFI [Benso98b, Benso98c] una herramienta SWIFI que utiliza las excepciones del procesador para realizar la inyección. A partir de EXFI se desarrolló **FlexFi** [Benso99b], que permite inyectar fallos mediante SWIFI utilizando tres métodos de inyección diferentes, uno de ellos híbrido [Benso98a]. Los fallos se inyectan a través de interrupciones provocadas por una placa externa. De esta manera se reduce la intrusividad, a costa de tener que desarrollar circuitería.

En [Amendola97, Impagliazzo03] se presenta **LIVE** (*Low-Intrusion Validation Environment*), un entorno de validación que integra la inyección física mediante *hardware* y *software* (lo que se ha denominado en puntos anteriores HWIFI y SWIFI). El modelo de fallos que se utiliza es la corrupción de registros, modificando cualquier registro accesible por *software*. La inyección basada en *software* perturba los registros directamente accesibles por *software*, mientras que para perturbar los registros mediante *hardware*, se perturban los *buses* suponiendo que afectan a las transferencias de los datos al pasar estos por los *buses*. La inyección del fallo se realiza de dos formas diferentes. Si está basada en *software*, se realizan mediante una pequeña rutina de interrupción. Si está basada en *hardware*, se realiza un forzado en los *buses*.

Para realizar la monitorización del sistema, se utiliza un analizador lógico, con lo que se evitan la intrusión del monitor *software*. El analizador lógico también es utilizado para sincronizar la inyección del fallo. Se pueden inyectar fallos transitorios y permanentes. La distribución de los fallos está basada en los resultados de la simulación VHDL de un modelo a nivel de puerta de un procesador RISC y en métodos de predicción de la fiabilidad.

NFTAPE [Stott00] surge en la Universidad de Illinois como una ampliación de FTAPE [Tsai95a, Tsai95b]. En este caso, se introduce el concepto de "inyector de fallos ligero" (del inglés *LigthWeight Fault Injector*, LWFI), que es el elemento más simple que puede inyectar fallos. Así, una herramienta es un conjunto de LWFI que se incorporan (conectan) a la herramienta a través de una interfaz predefinida y común. Esta herramienta está pensada para ser aplicada tanto a sistemas distribuidos como sistemas individuales. En [Stott00] se muestra una versión de NFTAPE que incluye un inyector HWIFI (a nivel de *pin*) y otro SWIFI (utilizando las capacidades de depuración del sistema). Sin embargo, los autores pretenden que en NFTAPE se puedan aplicar todas las técnicas físicas de inyección (simulación, HWIFI y SWIFI), considerando el mayor número posible de subtécnicas: dependientes del sistema, a nivel de *pin*, *scan-chain*, SWIFI mediante interrupciones, SWIFI utilizando las capacidades de depuración (*debugger-based*), SWIFI mediante manejadores (*drivers*), etc.

eXception [Santos01] está siendo desarrollada a partir de Xception [Carreira98]. Al igual que la anterior, pretende incorporar cualquier tipo de técnica de inyección HWIFI. La diferencia estriba en que la conexión de los inyectores individuales a eXception se lleva a cabo mediante

TCP/IP, por lo que el computador principal y los inyectores pueden estar en lugares distintos. En [Santos01] se presenta un prototipo incluyendo dos módulos de inyección HWIFI (uno a nivel de *pin* por forzado y uno mediante *Scan-Chain*), además del módulo de inyección mediante SWIFI.

GOOFI (*Generic Object-Oriented Fault Injection Tool*) [Aidemark01] ha sido desarrollada en Java, y utiliza bases de datos compatibles con SQL, lo que la hace altamente portable entre plataformas distintas. Permite inyectar fallos mediante *Scan-Chain* y SWIFI aplicada en tiempo de compilación (o antes de la ejecución, es decir, *pre-runtime SWIFI*). Al igual que las dos herramientas mostradas anteriormente, sus autores pretenden que con GOOFI sea posible inyectar fallos mediante el mayor número posible de técnicas basadas en simulación, HWIFI y SWIFI.

En [Vargas99b] se hace una primera aproximación de la aplicación de la prueba del *software* a la validación de diseños descritos en lenguajes de descripción de *hardware*. Para ello, adaptan la mutación débil (del inglés *weak mutation*) para comprobar la verificación de la prueba de un sistema generado mediante co–diseño y descrito en un lenguaje de descripción de *hardware*. En la misma línea, en [Ejlali03] se presenta **FITSEC** (*Fault Injection Tool based on Simulation and Emulation Co-operation*), un inyector de fallos híbrido que combina la inyección basada en simulación con la inyección basada en emulación. Esta herramienta puede trabajar con modelos descritos en VHDL y Verilog y permite inyectar fallos en diferentes niveles de abstracción. El método que utiliza es dividir el modelo entre el simulador y el emulador, por lo que habrá partes del circuito simuladas y otras emuladas. La división del código sigue dos reglas principales: las partes sintetizables del modelo se pasan al emulador y aquellos componentes con señales a monitorizar se deben quedar en el simulador. La inyección de los fallos se realiza mediante la inserción de perturbadores y mutantes, denominados FIU (*Fault Injector Units*), en puntos específicos del modelo. La comunicación entre el simulador y el emulador debe realizarse de tal manera que los resultados finales deben parecer resultados de la simulación de todo el sistema. Este trabajo es parecido al presentado en [Zarandi03], en el cual se utilizan modelos sintetizables y no sintetizables, aunque sólo se simulan. Este último trabajo se comenta en el punto 3.1.1.

2.4 Resumen y conclusiones

En este capítulo se ha definido el concepto de *Confiabilidad* como la propiedad de los sistemas informáticos que da idea de la confianza que los usuarios pueden depositar en el servicio que proporcionan. La Confiabilidad engloba tres elementos: los *atributos* de la Confiabilidad, los *impedimentos* que se oponen a su consecución y los *medios* que permiten contrarrestar el efecto de los impedimentos.

Los atributos son propiedades más concretas que permiten medir la calidad del servicio prestado en función de la aplicación del sistema: Disponibilidad, Fiabilidad, Seguridad–-Inocuidad, Confidencialidad, Mantenibilidad e Integridad.

Los impedimentos se oponen a la consecución de los atributos, reduciendo la Confiabilidad: la aparición de *fallos* en el sistema provoca *errores* que a su vez pueden desencadenar *averías*, que son comportamientos anómalos que hacen incumplir la función del sistema. Las formas en las que un sistema puede averiarse difiere según el punto de vista (el *dominio* en el que se incumple la función del sistema, la *percepción* de las averías por los usuarios y las *consecuencias* provocadas). También se pueden clasificar los sistemas en función de su comportamiento ante las averías.

Existen cuatro métodos que permiten alcanzar una elevada Confiabilidad: la *Prevención de fallos*, que está íntimamente ligada con las técnicas de diseño; la *Tolerancia a fallos*, que consiste en la introducción en el sistema de mecanismos que permitan cumplir con la función del sistema a pesar de la existencia de fallos; la *Eliminación de fallos*, que trata de reducir la presencia y la gravedad de los fallos; y la *Predicción de fallos*, que consiste en la estimación del número de fallos, su incidencia y sus consecuencias. La Predicción de fallos y la Eliminación de fallos están muy íntimamente ligadas entre sí, de manera que se pueden englobar bajo el término *Validación*.

Mediante la Validación se pueden medir algunos parámetros que caracterizan la respuesta del sistema ante la existencia de fallos: los Coeficientes (o Factores) de Cobertura de Detección y Recuperación de errores y las Latencias de Detección y Recuperación de errores. Estos parámetros están relacionados con los Mecanismos de Tolerancia a fallos.

La Validación se puede llevar a cabo de dos maneras: teórica y experimental. La primera es una Predicción de fallos sobre un modelo analítico del sistema. La segunda incluye tanto la Predicción de fallos como la Eliminación de fallos, aplicadas sobre un modelo de simulación o un prototipo.

La *Validación experimental* es muy importante ya que permite calcular los valores de parámetros como las Latencias y los Coeficientes de Cobertura de Detección de errores de una manera más sencilla que los métodos analíticos. La Validación experimental se puede realizar observando el comportamiento del sistema real en su entorno de trabajo (lo cual es muy difícil de conseguir, y más si se tiene en cuenta que las tasas reales de fallo son muy bajas, lo que requiere unos tiempos de observación muy elevados), o mediante la *Inyección de fallos*, consistente en la introducción deliberada de fallos en un modelo o prototipo del sistema.

En este capítulo también se ha realizado una clasificación de las diferentes técnicas de inyección de fallos: inyección mediante simulación, implementadas mediante *hardware* (o HWIFI) e implementadas mediante *software* (o SWIFI). En el caso de estas últimas, sólo se ha profundizado en las técnicas SWIFI, o de emulación de fallos, puesto que las técnicas de mutación pertenecen al ámbito de la prueba de *software*. De las técnicas descritas, se ha realizado una descripción de las características de cada una de ellas, se han mostrado algunas de sus subtécnicas más destacadas, y se han estudiado algunos de los trabajos publicados más relevantes en cada caso, haciendo mención especial a las herramientas realizadas. Dentro de cada técnica se han establecido también las principales ventajas e inconvenientes de cada una de ellas.

Por último, se ha expuesto la nueva tendencia que se está observando en estos últimos años, a la que se ha denominado *Inyección de fallos híbrida*. Esta nueva corriente intenta combinar las ventajas de diferentes técnicas de inyección en una única herramienta. La idea es permitir que sobre un mismo sistema se puedan aplicar varias técnicas de inyección que se complementen entre sí y permitan obtener unos resultados mucho más fiables y evitando las desventajas que surgirían si únicamente se utilizase una única técnica de inyección.

3 Técnicas de inyección de fallos basadas en VHDL

3.1 Introducción

En este capítulo van a describirse más detalladamente las diferentes técnicas de inyección de fallos basadas en VHDL, ya que centran la mayor parte del trabajo realizado en la presente tesis, pero sin entrar en detalles del lenguaje. Una explicación más detallada de las facilidades que ofrece el VHDL a la hora de inyectar fallos puede encontrarse en [DGil99a, Baraza03].

Del lenguaje en sí, podemos mencionar que el VHDL [IEEE93] se ha convertido en uno de los lenguajes más apropiados desde el punto de vista de la simulación de fallos. Las razones del extendido uso de VHDL se pueden resumir en:

- Es un lenguaje ampliamente utilizado en el diseño digital actual.
- Permite describir el sistema a distintos niveles de abstracción [Aylor90, Dewey92] (puerta, RT, chip, algorítmico, sistema, etc.)[14] gracias a la posibilidad de descripciones estructurales y comportamentales en un único elemento sintáctico.
- Ofrece muy buenas prestaciones en la modelización a alto nivel de sistemas digitales.
- Capacidad para soportar actividades de prueba [Miczo90].
- Algunos elementos de su semántica facilitan la inyección de fallos.

Para realizar la inyección de fallos sobre modelos en VHDL, en la bibliografía consultada se proponen básicamente dos grupos de técnicas, en función de si implican o no la modificación del código fuente del modelo [Arlat92a, Jenn92, Rimén92a, Jenn93a, Jenn93b].

El primer grupo de técnicas se basa en la utilización de las órdenes del simulador (en inglés *simulator commands*) para modificar el valor y la temporización de las señales y variables del modelo [Ohlsson92, Jenn94b, DGil98a, DGil99a, DGil00, Baraza00], sin tener que modificar el código VHDL del modelo. El segundo grupo se apoya en la modificación del código VHDL del modelo, mediante la inserción de componentes perturbadores (del inglés *saboteurs*) en arquitecturas estructurales [Boué96, Boué98, Amendola96, Folkesson98, Gracia01a], o creando mutaciones (del inglés *mutants*) de componentes ya existentes [Ghosh91, Armstrong92, Gracia01b, DGil03a, DGil03b].

Existe un tercer grupo de técnicas que se basa en la ampliación de los tipos y en la modificación de las funciones del lenguaje VHDL, a las que denominaremos *otras técnicas* [DeLong96a, Sieh97b]. La Figura 7 muestra una clasificación de las diferentes técnicas de inyección de fallos sobre modelos en VHDL.

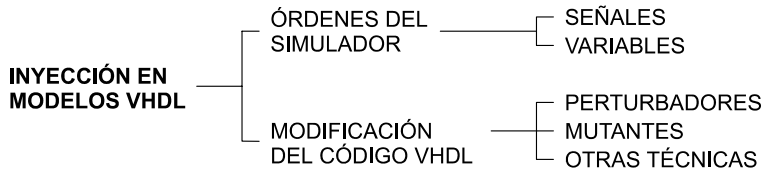


Figura 7. Clasificación de las técnicas de inyección sobre modelos VHDL [DGil99a].

14 Respecto a la clasificación de los diferentes niveles de abstracción, ésta se verá en el punto 4.3, "Modelos de fallos".

3.1.1 Antecedentes previos

Antes de comenzar a describir el trabajo realizado durante el desarrollo de la presente tesis, se van a describir y comentar algunas de las publicaciones más destacables relacionadas con la inyección de fallos mediante simulación de modelos en VHDL. Estas publicaciones incluyen tanto la descripción y/o aplicación de algunas herramientas así como la descripción de algunas propuestas para optimizar la utilización de las diferentes subtécnicas. En los puntos siguientes de este capítulo se explicarán en detalle las diferentes subtécnicas resumidas en la Figura 7.

MEFISTO (*Multi level Error and Fault Injection Simulation TOol*) [Rimén92a, Rimén93a, Jenn94b] quizás sea la herramienta más emblemática de esta técnica de inyección, debido principalmente a su carácter precursor. Esta herramienta surgió de la colaboración entre el LAAS-CNRS de Toulouse (Francia) y la Universidad de Chalmers (Göteborg, Suecia). En esta primera versión se establecieron las bases para realizar la inyección de fallos en VHDL con distintas técnicas, aunque únicamente implementaron la inyección de fallos basada en comandos del simulador. Después de esta herramienta, surgen dos versiones más evolucionadas de MEFISTO, que son **MEFISTO-C** y **MEFISTO-L** [Boué97, Rimén97, Boué98, Folkesson98, Arlat03a]. Ambas herramientas utilizan como simulador el *Vantage Optium VHDL Simulator*, realizando dicha simulación en paralelo, en una red de estaciones de trabajo *Sun IPC* bajo UNIX, lo cual permite realizar la inyección de los fallos de manera distribuida.

MEFISTO-C [Rimén97, Folkesson98, Arlat03a] inyecta fallos transitorios y permanentes en variables y señales del modelo VHDL mediante comandos del simulador, y aunque no puede operar con perturbadores, aseguran que éstos pueden ser fácilmente tratados con la herramienta. Se puede aplicar a modelos realizados a nivel de puerta y RT. Los modelos de fallos son los mismos que los inyectados por MEFISTO, los cuales se verán más adelante.

Por su parte, MEFISTO-L [Boué98, Arlat03a] inyecta fallos añadiendo perturbadores al modelo, teniendo como principal característica que cada perturbador sólo implementa un modelo de fallo. Estos modelos son *stuck-at* (‘0’, ‘1’), indeterminación, alta impedancia (*open-line*) y *bit* o *value-flip*. MEFISTO-L también añade sondas al modelo para poder observar si se cumplen las aserciones de los mecanismos tolerantes a fallos. Otro aspecto muy destacable de MEFISTO-L es el hecho de que es independiente del simulador VHDL utilizado, ya que además de los perturbadores que se insertan en el modelo, también se introducen unos elementos especiales, llamados sondas (en inglés, *probes*), que son componentes similares a los perturbadores, pero cuya misión no es alterar el valor de una señal del modelo, sino leerlo, sea para realizar tareas de control (por parte del módulo de inyección) o para la observación de la evolución del circuito tras la inyección de fallos.

El funcionamiento de todas las herramientas es similar, por lo que nos centraremos en MEFISTO, al ser la primera de ellas. MEFISTO consta de tres partes principales: Generación de experimentos (*Setup Phase*), Simulación (*Simulation*) y Procesamiento de los datos recogidos (*Data Processing Phase*), tal y como se puede ver en la Figura 8.

Respeto a los modelos de fallos, MEFISTO puede inyectar fallos transitorios y permanentes en señales y variables de modelos comportamentales y estructurales realizados a nivel de puerta o RT. Los modelos de fallos pueden ser predefinidos (como los *stuck-at* ‘0’ y ‘1’ y *bit-flip* asociados a los tipos de datos de tipo *bit*[15]) o definidos por el usuario. Todos los modelos están definidos a partir del concepto de *modelos abstractos de fallos* (o AFM, del inglés *Abstract Fault Models*).

Un aspecto interesante de MEFISTO es el hecho de que el instante de inyección se puede generar de tres maneras: (i) mediante la especificación de un valor de tiempo (fijo o generado aleatoriamente mediante una función de distribución uniforme), (ii) indicando un patrón de valores en las señales del modelo, y (iii) mediante el uso de puntos de ruptura (*breakpoints*). En

[15] Esta denominación incluye todos los tipos de datos estándar que se pueden aplicar a elementos que se corresponden con líneas físicas a nivel de puerta: `bit`, `std_ulogic`, `std_logic`, etc.

los dos últimos casos, además, se puede establecer un número determinado de ocurrencias de la condición buscada.

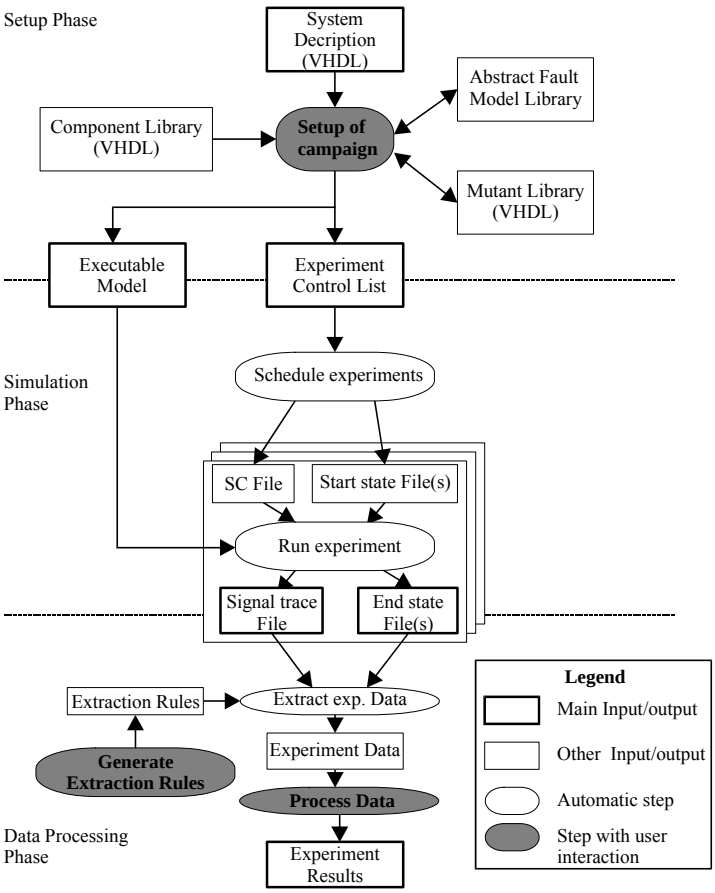


Figura 8. Estructura general de la herramienta MEFISTO [Jenn94b].

VERIFY (*VHDL-based Evaluation of Reliability by Injecting Faults EfficientlY*) [Sieh96, Sieh97a, Sieh97b, Sieh97c] es una herramienta de inyección de fallos en VHDL que permite inyectar fallos permanentes y transitorios en sistemas digitales modelados en VHDL. La metodología que siguen es la de modificar el código VHDL para incluir las descripciones del comportamiento de los fallos en el propio modelo VHDL. Esta descripción de los fallos incluye la duración del fallo así como el tiempo medio entre fallos[16] [Sieh98]. Es decir, según la clasificación de la Figura 7, se correspondería con el ítem "*Otras técnicas*".

La descripción comportamental del componente debe ser modificada para que ejecute las acciones pertinentes en caso de que un fallo se active. Los autores hacen notar que los parámetros de los fallos (frecuencia, duración, etc.) pueden ser fácilmente ajustables al entorno en el que el sistema será utilizado gracias al intercambio de las librerías básicas, aprovechando además el conocimiento de los fallos que los proveedores de *hardware* (y por tanto de las librerías) poseen de los mismos.

Las señales de inyección de fallos las denominan FIS (del inglés *Fault Injection Signals*). Con el fin de mantener cada FIS de cada módulo independiente y transparente al resto de módulos, introducen un nuevo tipo de señal en la sintaxis del VHDL.

Para poder utilizar esta extensión de las sintaxis del VHDL, han desarrollado un compilador que extrae los datos de los FIS, así como un simulador que automatiza la inyección de fallos y varias herramientas para el análisis de resultados. En la Figura 9 se puede ver un diagrama de bloques de las diferentes fases y módulos de la herramienta. En [Sieh97a] se presenta la

[16] En inglés *Mean Time Between Failures* o MTBF.

aplicación de VERIFY sobre una versión reducida del procesador DP32 [Ashenden92], al que se inyectan fallos *stuck-at* (‘0’, ‘1’) (en los *pines* y en la circuitería combinacional) y *bit-flip* (en los registros internos).

Para acelerar el tiempo de simulación, utilizan una técnica denominada *multi-threaded fault injection* [Güthoff95]. Con esta técnica sólo ejecutan una simulación completa y sin fallos (denominada *golden run*). Durante esta simulación, se guardan puntos de control en ciertos instantes de tiempo. Estos puntos de control son utilizados después para acortar la simulación cuando se ha inyectado un fallo. La simulación se para si la simulación con fallo alcanza un estado idéntico a la simulación sin fallo (el fallo se ha recuperado) o el intervalo de simulación del fallo ha alcanzado un cierto límite (*timeout*). Otro método que utilizan para ahorrar tiempo en la simulación es hacer un estudio previo del sistema e inyectar fallos en aquellos puntos que más afectan al sistema, no inyectando en aquellos puntos en los que el fallo no vaya a producir ningún efecto.

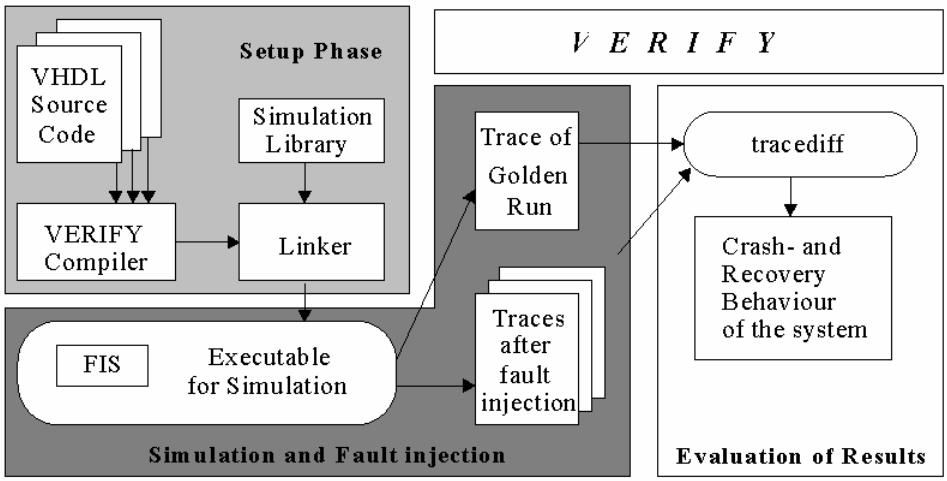


Figura 9. Estructura global de la herramienta de inyección VERIFY [Sieh97b].

En [DeLong94, DeLong96a] se presenta otra propuesta para la realización de inyección de fallos basada en simulación, proponiendo la inyección en modelos comportamentales del sistema, denominados modelos ISA (del inglés *Instruction Set Architecture*). Estos modelos pueden ejecutar código máquina, contando con la ventaja añadida que las simulaciones son mucho más rápidas que utilizando otro tipo de modelos.

En una segunda aproximación [Ghosh95a, Ghosh95b] amplían la propuesta de inyección en modelos comportamentales a modelos que representan múltiples niveles, incluyendo modelos sintetizables. En este caso distinguen la inyección en modelos de alto nivel de abstracción y modelos de bajo nivel de abstracción.

En modelos de alto nivel, la inyección de fallos se realiza instanciando componentes dedicados exclusivamente a la inyección de fallos (denominados FIM, del inglés *Fault Injection Modules*). La inyección se realiza en el flujo de información del modelo. Para ello, los FIM corrompen el flujo de datos alterando, creando o destruyendo información. El modelo de fallos usado en este nivel es una manifestación del efecto de los fallos a nivel de puerta que afectan a la información que fluye por el modelo.

En modelos de bajo nivel, la idea básica para la inyección de fallos es similar a la inyección en modelos de alto nivel mediante la instanciación de un FIM, el cual corrompe la señal elegida de acuerdo con la información del fallo proporcionada por el diseñador. El modelo de fallos utilizado en este nivel es el *stuck-at* (‘0’, ‘1’).

La técnica que se propone en [DeLong96a] es la extensión de los tipos de datos para que contengan información accesible por la función de resolución del *bus* y que sea ésta la que

inyecte los fallos. Así pues, además de modificar o crear una nueva función de resolución del *bus*, se necesita añadir un pequeño proceso de inyección en los módulos VHDL, el cual lee los datos de la inyección desde un fichero de texto.

Para la elaboración de los modelos en VHDL de los diferentes sistemas, y para su simulación tanto en modo normal como inyectando fallos, los autores han utilizado ADEPT [Kumar94, Ghosh95c], una herramienta de inyección de fallos comentada anteriormente (concretamente, en el punto 2.2.4.1.5).

En [Amendola96], más que un entorno para la inyección y simulación de fallos en VHDL, lo que se hace es un análisis específico de un sistema microprocesador. Para ello, en primer lugar implementan un modelo en VHDL al nivel RT de un microprocesador para comunicaciones en tiempo real, al que después le añaden perturbadores específicamente creados para este sistema o modifican ciertas partes del mismo para la inyección de fallos. Después de la inyección, analizan los datos con un programa también específicamente creado para este experimento. El modelo de fallos utilizados es el *bit-flip*, e inyectan fallos en la memoria, en los registros internos del procesador y en los *buses*.

En el caso de [Riesgo96] se crean vectores de prueba para la fabricación de circuitos integrados. En especial, crean modelos de fallos para sistemas descritos a nivel de registros en VHDL presintetizable. El modelo de fallos sólo considera aquellos fallos fácilmente traducibles en fallos *hardware*. Los fallos son inyectados mediante mutantes (la técnica de los mutantes se explicará en el punto 3.2.2.2), creando una arquitectura por fallo. Este modelo de fallos está basado en fallos simples, permanentes y de ámbito local, y se divide en tres clases:

- Fallos en los datos (señales y variables): modelo de *stuck-at*.
- Fallos en expresiones: también utilizan el modelo de *stuck-at*. Identifican dos tipos de expresiones: datos y control.
- Fallos en sentencias (en inglés *dead sentences*): las sentencias afectadas no se ejecutan.

En [Celeiro96] se muestra la creación de una metodología para derivar modelos de fallos para la prueba de circuitos integrados CMOS, soportando el VHDL en este caso, un modelado, una inyección y una simulación realista de los fallos, creando para ello un conjunto de herramientas específicas. Asumen que los fallos realistas son el puenteo (del inglés *bridging*), puesto que es el fallo más común en tecnología CMOS. Los fallos son descritos en VHDL mediante un modelo comportamental, añadiendo esta descripción de los fallos al sistema por medio de perturbadores (la técnica de los perturbadores se explicará en el punto 3.2.2.1). La simulación del sistema se lleva a cabo creando un modelo del sistema para cada inyección.

En [DeLong96b] utilizan el estándar IEEE 1029.1–1996[17] para la simulación de fallos. Con este método realizan la inyección del fallo independientemente del simulador. El procedimiento para la inyección del fallo es la inserción en el banco de pruebas del circuito de un perturbador (un proceso WAVES), el cual inyecta el fallo. A partir de este trabajo, en [Hayne99] se proponen nuevos modelos de fallos para descripciones comportamentales en VHDL de circuitos lógicos combinacionales a partir del modelo de *stuck-at* en una línea.

En [Coppens98] utilizan el VHDL para modelar sistemas a nivel de puertas, con librerías basadas en tecnología Nortel's 0.8 μm BiCMOS. Una vez que tienen el modelo en VHDL, le añaden mutantes y perturbadores, los cuales modelan fallos a bajo nivel.

En [López98a, López98b] se presenta una herramienta de simulación de errores utilizada para estimar la calidad de los bancos de pruebas usados para validar diseños VHDL. Estos diseños están descritos en VHDL sintetizable. El método está basado en la detección de ciertos tipos de errores, fundamentando este modelo de errores en tres clases, las cuales representan los

[17] Este estándar se denomina WAVES (Waveform and Vector Exchange). Define cómo aplicar vectores de prueba a modelos VHDL.

errores más comunes. Estos errores son utilizados como referencia para implementar los distintos mutantes.

En [Al-Hayek99] se adapta la prueba de mutación del *software* a la prueba de circuitos descritos en VHDL funcional. Utiliza métricas estándar de la prueba de *software* para calcular la calidad del proceso de validación del diseño. También se emplean coberturas de fallos *hardware* para calcular la calidad de la prueba en el nivel del *hardware*. Para realiza todo esto utiliza mutantes creados mediante el cambio de un operador por otro, inyectando fallos permanentes del tipo *stuck-at* y *stuck-open*. Continuando este trabajo, en [Robach03] introducen nuevos modelos de fallos funcionales en el dominio del *hardware* y se presenta **ALIEN**, un entorno diseñado para realizar la prueba de mutación de diseños descritos en VHDL funcional. ALIEN tiene tres funciones principales: genera los mutantes, genera los datos para la prueba y compara las simulaciones del diseño original con el diseño mutado.

En [Vargas00a] presentan una herramienta CAD denominada **FT-PRO**. Esta herramienta genera elementos de memoria tolerantes a fallos transitorios, en especial SEU. Una vez generado el circuito con esta herramienta, inyectan fallos modificando el código, creando una especie de mutante que inyecta el fallo (en este caso *bit-flips*) según un MTBF determinado previamente.

En [Corno00] se presenta **Fault Detection System**, un prototipo de herramienta que aplica la técnica de las órdenes del simulador a modelos de nivel RT. La herramienta funciona sobre una estación de trabajo *Sun Ultra* 5 bajo UNIX. El control del instante de inyección se realiza mediante puntos de ruptura asociados a sentencias. El prototipo mostrado sólo permite inyectar fallos permanentes, y los únicos modelos de fallo inyectados son *stuck-at* '0' y '1'.

En [Shaw01, Shaw03] se presenta un modelo de perturbador para inyectar fallos de tipo *bridging* en modelos estructurales a nivel de puerta. El modelo mostrado se basa en redes neuronales MLFN (del inglés *multilevel feedforward neural network*), por lo que a las entradas habituales del perturbador (señales procedentes de la salida de puertas lógicas) se les añaden las entradas de los circuitos (puertas) que las generan. Otro aspecto interesante es el hecho de que la entrada de control del perturbador no sólo distingue entre la existencia o no del fallo, sino también entre fallos "duros" (*hard*) y "blandos" (*soft*). La diferencia entre ambas clases de fallo está relacionada con la resistencia eléctrica del cortocircuito producido. Así, una baja resistencia da lugar a fallos duros, que provocan fallos lógicos, mientras que una resistencia elevada produce fallos blandos, que originan una degradación de la operatividad (el efecto puede ser por ejemplo una alteración de los retardos).

En [Cardarilli02] se muestran perturbadores que se aplican a un diseño comportamental. En este caso, el perturbador se ha diseñado como un proceso adicional que se añade al código comportamental, en vez de diseñarlo como un componente, y se emplea para alterar algunas partes del modelo. El perturbador propuesto permite inyectar fallos de tipo *bit-flip* en los registros de un modelo comportamental de un microcontrolador 80C51.

En [Zarandi03] se presenta **SINJECT**, una herramienta capaz de inyectar fallos en modelos descritos tanto en VHDL como en Verilog. Esta herramienta intenta aprovechar las ventajas de ambos lenguajes, siendo estas ventajas según los autores que el lenguaje VHDL es más adecuado para el modelado en altos niveles de abstracción, mientras que el lenguaje Verilog es más realista en las simulaciones. Para realizar la inyección utilizan perturbadores en Verilog y perturbadores y mutantes en VHDL. En Verilog, los perturbadores pueden insertarse en todos los niveles de abstracción, mientras que en VHDL emplea únicamente dos modelos: el perturbador serie simple y el perturbador paralelo. Para probar SINJECT realizan una campaña de inyección en le procesador DP32, descrito en VHDL, y otra campaña en el procesador aritmético ARP, descrito en Verilog.

Existen dos campos más donde se aplican las técnicas de inyección de fallos basada en VHDL. El primero es el de la prueba (o *test*), donde uno de los objetivos principales es la reducción del número de patrones de prueba que hay que aplicar a un sistema sin que se reduzcan la validez estadística de los resultados obtenidos, lo que se denomina *Fault List*

Collapsing, o simplemente *Fault Collapsing* [Smith96, Aftabjahani97, Benso98d]. El segundo campo es la inyección mediante emulación con FPGA. La técnica más frecuentemente utilizada es la de mutantes (sobre descripciones en los niveles de puerta o RT). En este caso, los mutantes se aplican, no para su simulación, sino para incluirlos en un prototipo sintetizable en una FPGA. La mutación de un diseño se puede realizar de dos maneras:

- Modificando algunas sentencias de asignación (de registros, *buses* o máquinas de estados) para inducir valores erróneos. La inyección se suele llevar a cabo mediante funciones que asignan el valor correcto en caso de no inyectar fallos, o un valor incorrecto (entre diferentes posibilidades), cuando se inyecta un fallo [Leveugle00b].
- Modificando la descripción a nivel de puerta del diseño, en particular, los biestables del modelo, que se reemplazan por otros que son, a su vez, un diseño estructural que implementa un mecanismo similar al *Scan-Chain* (véase el apartado 2.2.2.2.3) [Civera01c, Velazco01a, Velazco01b].

La inyección de fallos en [Leveugle00b] se realiza mediante la generación de mutantes sintetizables que cambian estados de una máquina de estados finitos. Los fallos se inyectan en la entrada del registro de estado de esta máquina de estados finitos, o en elementos similares generados a partir de un mapa de flujo de control en el nivel RT. El modelo de fallo representa *bit-flips* causados por SEU en partes de control.

En [Velazco01a, Civera01a] se utilizan también los mutantes mediante métodos similares. La técnica se aplica a modelos estructurales a nivel de puerta. La mutación consiste en reemplazar los biestables originales por otros capaces de inyectar fallos, teniendo en cuenta que el número de mutantes que se pueden añadir al modelo depende de la capacidad de la FPGA utilizada para simular el diseño. De este modo, es probable que la aplicación de todos los mutantes tenga que realizarse por "tandas", de manera que en cada "tanda" se incluyen tantos mutantes como quepan en la FPGA. En lo que se refiere a los mutantes propiamente dichos, el método propuesto está pensado inicialmente para ser aplicado a modelos comportamentales, aunque es posible pensar en la mutación de modelos estructurales.

Mención aparte merecen los trabajos presentados en [Parrotta00a, Parrota00b, Berrojo02, Corno03]. En [Parrotta00a, Parrota00b] intentan disminuir el tiempo de inyección y simulación del modelo. Para ello, utilizan dos tipos de análisis que denominan estático y dinámico. Durante el análisis estático, realizado antes de comenzar las simulaciones, eliminan aquellas inyecciones cuyos efectos pueden ser evaluados a priori. El análisis dinámico se realiza mientras se ejecutan las simulaciones. Para ello comparan el estado de la simulación actual (simulación con un fallo inyectado) con una simulación sin fallos. Tan pronto como la simulación con fallos alcanza un estado conocido dentro de la simulación sin fallos, y por tanto, se sabe cual va a ser el efecto del fallo, detienen la simulación con fallos[18]. Para probar estas técnicas, han desarrollado una herramienta de inyección de fallos que utilizando la técnica de órdenes del simulador, combinada con puntos de ruptura, inyecta fallos transitorios de tipo *bit-flip* sobre el modelo de un microprocesador i8051 ejecutando distintos programas de prueba. Las inyecciones se realizan en memoria y en registros accesibles por el usuario.

Por su parte, en [Berrojo02, Corno03] también se utiliza la técnica de órdenes del simulador (combinada con el uso de puntos de comprobación/recuperación para acelerar la simulación) para inyectar fallos transitorios de tipo *bit-flip*. La técnica propuesta se aplica sobre el modelo en VHDL de un dispositivo utilizado para el control de satélites (control de los paneles solares, telemetría, etc.) llamado SADE (del inglés *Solar Array Drive Electronics*), fabricado por *Alcatel Espacio*. Además de utilizar el análisis estático y dinámico de [Parrotta00a, Parrota00b], utilizan también mecanismos para reducir el tiempo de simulación de cada fallo, como la restauración

[18] La simulación sin fallos puede ser vista como un conjunto de estados. Cuando la simulación con fallos alcanza un estado de este conjunto, los autores afirman que el resto de la simulación con fallos será idéntica a la simulación sin fallos, deteniéndo así la simulación con fallos..

del estado del sistema justo antes de realizar la inyección, ahorrando el tiempo de simulación anterior a la inyección.

3.2 Técnicas de inyección de fallos basadas en VHDL

En este punto se va a profundizar en el estudio de las diferentes técnicas de inyección de fallos basadas en VHDL, así como se realizará una amplia descripción de las aportaciones realizadas durante el desarrollo de la presente tesis.

Si recordamos la Figura 7, las diferentes técnicas de inyección se podían clasificar en función de si se manipulaba el modelo VHDL o no. Si no se manipula el modelo, los fallos se inyectan mediante las órdenes del simulador, que a su vez presenta dos variantes: manipulación de señales y manipulación de variables.

3.2.1 Inyección de fallos mediante órdenes del simulador

En la mayoría de los simuladores VHDL, existen algunas órdenes o comandos del simulador que permiten la modificación del valor de las señales y de las variables del modelo durante la simulación del mismo[19]. Utilizando adecuadamente estas órdenes, y asignando ciertos valores durante un tiempo determinado, se puede realizar la inyección de fallos. Dentro de esta técnica se pueden distinguir la manipulación de señales de la manipulación de variables, tal y como se estudia a continuación.

3.2.1.1 Manipulación de señales

Mediante esta técnica, los fallos se inyectan alterando el valor o las características temporales de las señales del modelo VHDL. La inyección se realiza desconectando la señal de su *driver* o *drivers* para después forzar el nuevo valor. Una vez finalizada la inyección, la señal se vuelve a conectar a su *driver(s)*. El algoritmo a realizar sería el siguiente [DGil99a, DGil99b, Baraza00]:

1. Simular hasta el instante de inyección.
2. Desconectar el(los) *driver(s)* de la señal y asignar el valor modificado (valor inyectado).
3. Simular durante el tiempo de duración del fallo.
4. Volver a conectar la señal a su(s) *driver(s)*.
5. Simular hasta finalizar el tiempo de observación.

Esta secuencia sería la utilizada para inyectar fallos transitorios, que son los fallos más comunes y los más difíciles de detectar [Iyer86, DBENCH02]. Si lo que se pretende es inyectar un fallo permanente, se deben omitir los pasos 3 y 4, mientras que repitiendo los pasos 1 al 4 con intervalos de tiempo separados aleatoriamente, se podrían inyectar fallos intermitentes.

3.2.1.2 Manipulación de variables

En este caso, es posible alterar los valores de las variables del modelo, lo cual permite la inyección de fallos en modelos comportamentales de los componentes. Los pasos a seguir serían [DGil00, Gracia00a, DGil03a]:

1. Simular hasta el instante de inyección.

[19] Este es el caso del simulador *Modelsim* de *Mentor Graphics*, que es el utilizado por el GSTF [Model98, Model01a].

2. Asignar a la variable el nuevo valor.
3. Simular hasta que finalice el tiempo de simulación.

A pesar de que la operación es muy similar a la manipulación de señales, en este tipo de inyección es muy importante tener en cuenta que no es posible controlar la duración de la inyección, con lo que no se pueden inyectar fallos permanentes. Esto es debido a que las variables no pueden ser desconectadas de su *driver*, si no que solamente se les puede asignar un nuevo valor, que eventualmente, puede ser modificado durante el tiempo restante de observación de la simulación del sistema.

3.2.1.3 Modelos de fallos

Hay que comentar, aunque sea de forma breve, los posibles modelos de fallos que se pueden inyectar con esta técnica. Gracias a las capacidades del simulador comercial utilizado [Model98, Model01a], se han podido inyectar diferentes modelos de fallos, sobrepasando los clásicos modelos de *bit-flip* para fallos transitorios y *stuck-at* para fallos permanentes. Estos modelos de fallos han sido ampliados con los modelos *open-line*, inversión lógica (*bit-flip* en elementos de memoria y *pulse* en elementos combinacionales), *indetermination* y alteración de los retardos de propagación (más conocido como *delay*). Además, estos fallos pueden ser inyectados con distintas duraciones: transitorios, permanentes e intermitentes. En el punto 4.3, "Modelos de fallos", se describirán con mayor detalle el significado y los mecanismos físicos relacionados con estos modelos de fallos.

3.2.2 Inyección de fallos mediante la modificación del modelo VHDL

A continuación, se presentarán dos técnicas que mediante la manipulación y/o modificación del modelo VHDL inyectan los fallos. Los elementos utilizados en la primera técnica se denominan *perturbadores* [Boué98, Arlat99, Gracia01a, Gracia01b, DGil03b]. Estos elementos se añaden al modelo como componentes nuevos, y son utilizados para inyectar fallos en modelos estructurales. La segunda técnica utiliza *mutantes* para inyectar fallos en modelos comportamentales [Jenn94b, Rimén97, Gracia01a, Gracia01b, DGil03b]. Estos componentes se basan en la alteración de componentes ya existentes en el modelo VHDL.

3.2.2.1 Perturbadores

Un perturbador se puede definir como un componente que se añade al modelo VHDL con la función de alterar el valor o las características temporales de una o más señales a la hora de inyectar un fallo. La inyección se realiza cuando el perturbador es activado, permaneciendo éste inactivo durante el funcionamiento normal del sistema. Hay que tener en cuenta que las señales donde los perturbadores inyectan fallos son aquellas que conectan componentes en modelos estructurales. La arquitectura interna de los perturbadores puede ser comportamental o estructural. En el primer caso, el perturbador está compuesto básicamente por un proceso que se activa con los cambios en las señales de control, entrada y/o salida. En el segundo caso, la arquitectura se puede realizar mediante multiplexores [DGil99a]. Existen dos tipos de perturbadores básicos:

a) *Perturbador serie*. En este caso, el perturbador rompe la conexión entre la salida de uno o varios módulos y las correspondientes entradas en otro módulo(s) [Jenn93b, DGil99a]. Pueden ser simples, cuando el número de entradas y salidas es 1, o complejos, cuando alguno de ellos es mayor de 1, como se pueden ver en la Figura 10 y la Figura 11. El fallo es inyectado cuando la señal *Control* es activada.

b) *Perturbador paralelo*. Se consigue añadiendo un *driver* adicional a una señal asociada a una función de resolución [Jenn93b, DGil99a], tal y como se puede ver en la Figura 12. Como en el caso anterior, al ser activada la señal *Control*, se inyecta el fallo.

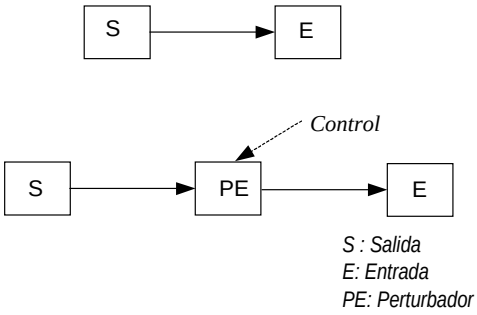


Figura 10. Estructura del Perturbador Serie Simple.

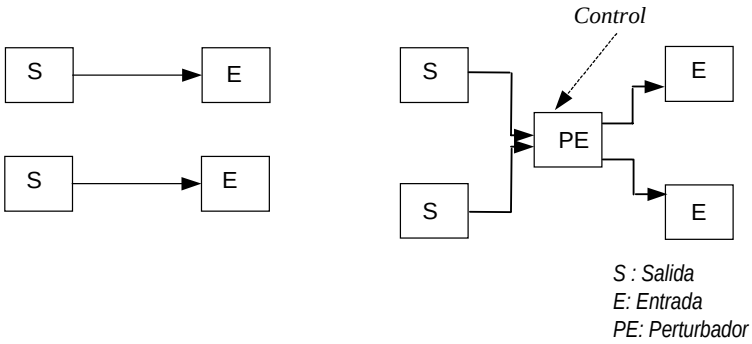


Figura 11. Estructura del Perturbador Serie Complejo.

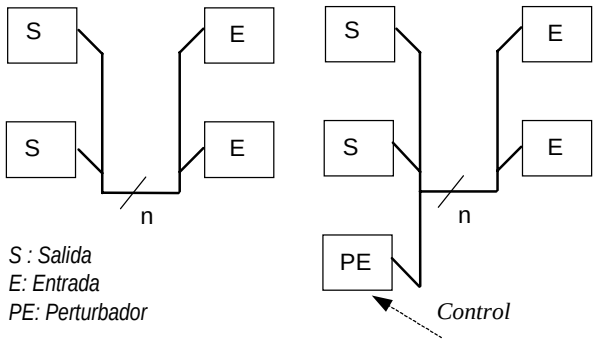


Figura 12. Estructura del Perturbador Paralelo.

Durante el trabajo realizado en esta tesis, a partir de los perturbadores básicos mostrados en la Figura 10, la Figura 11 y la Figura 12, se han desarrollado nuevos tipos de perturbadores con la idea de una máxima extensión de la inyección de fallos en el sistema bajo prueba [Gracia01a]. Hay que destacar también la ampliación realizada en el diseño de algunos de los perturbadores con el fin de poder inyectar fallos en *buses*.

Un aspecto importante en la inyección de los fallos mediante perturbadores es la activación o sincronización del mismo. Al activarse la señal de control, se produce un evento, como puede ser un flanco de subida o de bajada. Dependiendo de la sincronización de la inyección del fallo con la señal de control, se puede producir más de una inyección por cada activación de esta señal de control. Para realizar una única inyección, ésta debe estar sincronizada con el flanco de subida (o bajada) de la señal de control, tal y como se puede ver en la Figura 13-a. Si se realiza una sincronización por nivel (Figura 13-b) se podría producir más de una inyección, debidas a los eventos generados con cada activación de la señal de entrada (*I*).

La Figura 13 muestra en detalle el diagrama temporal de la activación del fallo por flanco o por nivel. En este caso, se muestra la inyección de fallos utilizando un perturbador serie simple, cuyo proceso se activa con la ocurrencia de un evento en las señales de entrada y de control, tal

y como se verá en detalle posteriormente. La Figura 13-a muestra la activación de la inyección por flanco. En este caso, y tal y como se ha comentado anteriormente, al activarse la señal de control se realiza la inyección. Al estar ésta sincronizada con este evento (flanco de subida en la señal de control), sólo se inyecta un fallo. En cambio, en la Figura 13-b se puede ver la activación de la inyección del fallo por nivel. Como se puede ver, el problema surge con el resto de eventos que se producen en la señal de entrada, los cuales activan ese proceso. Mientras la señal de control esté a nivel alto, cada vez que se produzca un evento en ese proceso (cambios de valor en la señal de entrada), se inyectará el fallo. Esto puede provocar cambios múltiples no deseados cuando se inyectan fallos en los que el nuevo valor depende de la entrada. Tal es el caso, por ejemplo, de los fallos que implican inversión, como el *bit-flip* y el *pulse*, así como en los fallos de tipo *delay*.

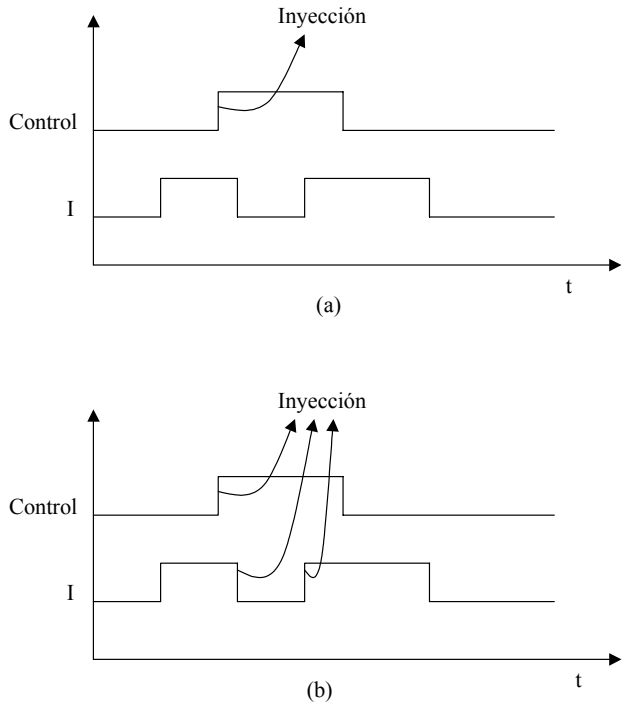


Figura 13. Temporización en la inyección de fallos. a) Inyección por flanco. b) Inyección por nivel.

Estas dos inyecciones se pueden distinguir fácilmente en VHDL gracias al atributo 'stable' de las señales. En el pseudo código siguiente se muestra la estructura básica de los procesos para la inyección de fallos por flanco o por nivel.

Inyección por flanco	Inyección por nivel

process (I, Control)
begin
if Control = '1' **and not** Control**'stable then**
selección_tipo_fallo;
inyección();
else
procesamiento_normal();
end if;
end process;
	process (I, Control)
	begin
	if Control = '1' **then**
	selección_tipo_fallo;
	inyección();
	else
	procesamiento_normal();
	end if;
	end process;

Un aspecto importante de esta técnica de inyección que también hay que tener en cuenta es la ubicación de los perturbadores. La Figura 14 muestra esta problemática. Si tenemos la arquitectura estructural de la Figura 14–a, se pueden insertar perturbadores en tres lugares, siendo los resultados obtenidos completamente diferentes:

- Si el perturbador se aplica en la salida del módulo "O" (Figura 14–b) se estaría simulando la existencia de fallos en la salida de este componente, los cuales afectarían por igual a todos los módulos que la recibieran como entrada.
- Si el perturbador se emplaza en la entrada del módulo "I1" (Figura 14–c), los fallos sólo afectarían a este módulo.
- Si el perturbador se sitúa en la entrada del módulo "I2" (Figura 14–d), de manera análoga al caso anterior, los fallos sólo afectan a este componente.

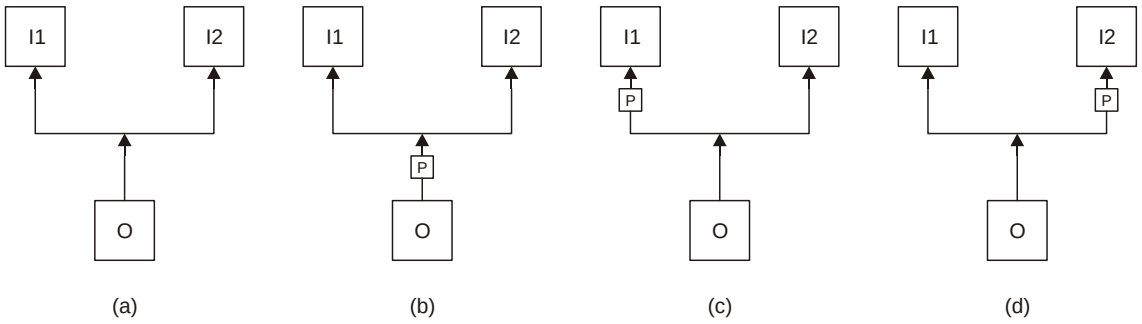


Figura 14. Estrategia de inserción de perturbadores [Boué96].

En cuanto a los perturbadores paralelos, en [DGil99a] se proponen dos métodos para implementar y realizar la inyección de forma efectiva:

- Modificar la función de resolución asociada a la señal a la que afectará el perturbador, con el fin de introducir el efecto prioritario del perturbador sobre el resultado de la resolución. Tal y como se definen las funciones de resolución, con este método es imposible que la función modificada pueda discernir la fuente de cada valor y, por lo tanto, identificar el valor prioritario correspondiente al perturbador. Para solucionar esta ambigüedad, habría que declarar un nuevo tipo de dato resuelto, que incluyera además de los valores lógicos de este tipo, unos nuevos valores que se correspondieran con los ya existentes, pero considerados "prioritarios". Estos nuevos valores "prioritarios" sólo deberían ser utilizados por el perturbador. De este modo, al especificar la función de resolución el nuevo tipo, al detectar la presencia de uno de los valores "prioritarios" el valor resuelto sería el valor "no prioritario" equivalente. Sin embargo, esta técnica podría ser englobada en la Figura 7 dentro del apartado "*Otras técnicas*", quedando fuera del ámbito de esta tesis.
- Introducir un componente adicional que recibiera como entradas las diferentes fuentes de la señal afectada, además de la salida del perturbador. Esta idea propone básicamente desconectar la señal de sus fuentes, por lo que la implementación es realmente la de un perturbador serie.

En [Gracia01b] se han propuesto e implementado diferentes extensiones a los perturbadores base de la Figura 10 y la Figura 11, los cuales se pueden ver en la Figura 15. Dichos modelos se aplican a señales bidireccionales y a *buses*, ampliándose incluso los modelos de fallos que se podían inyectar con los perturbadores base. El diseño de estos nuevos perturbadores, así como los nuevos modelos de fallos que se pueden inyectar se van a explicar en los puntos siguientes.

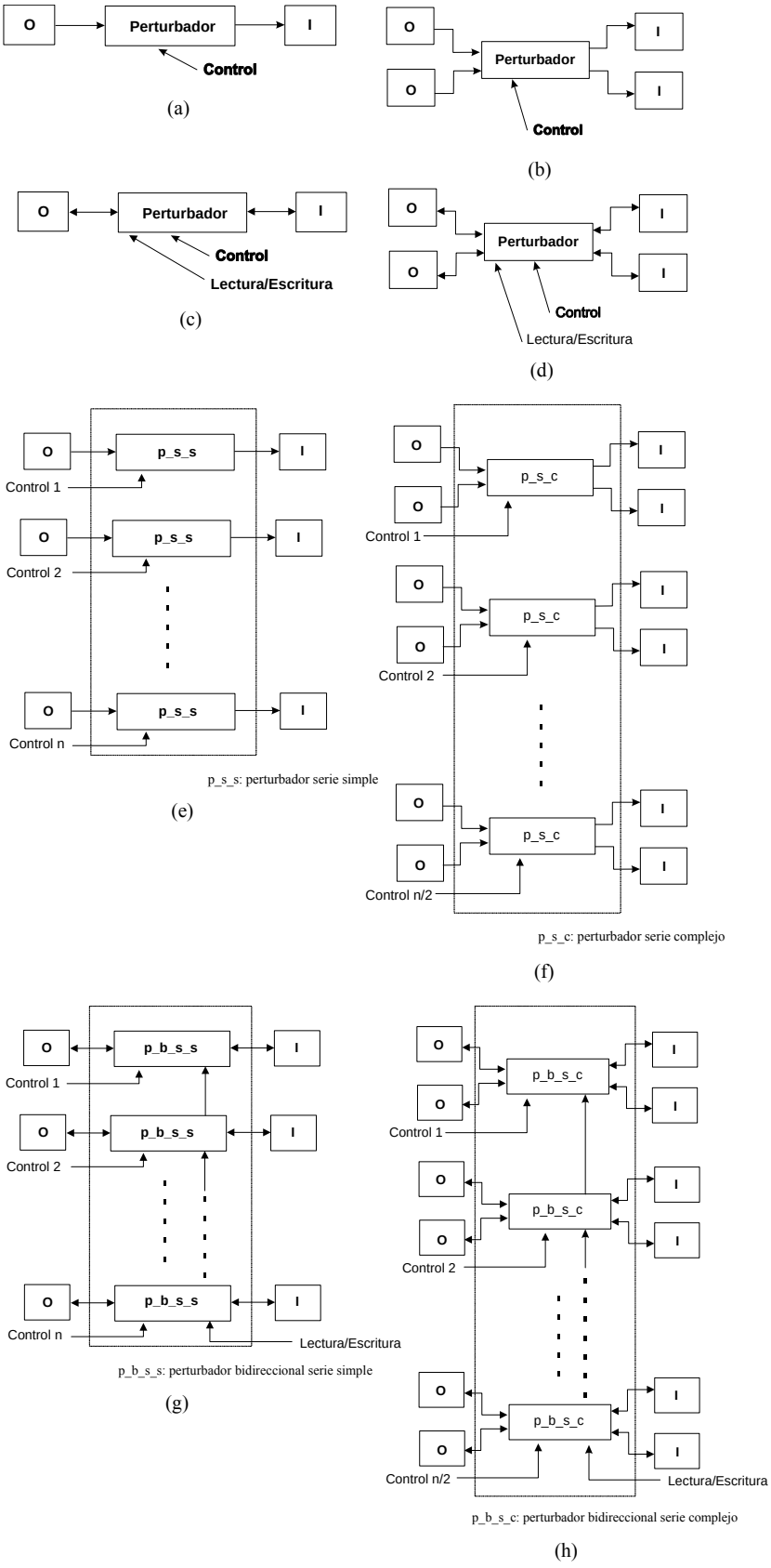


Figura 15. Tipos de perturbadores diseñados [Gracia01b]: (a) Perturbador Serie Simple; (b) Perturbador Serie Complejo; (c) Perturbador Bidireccional Serie Simple; (d) Perturbador Bidireccional Serie Complejo; (e) Perturbador Unidireccional Simple de *n* bits; (f) Perturbador Unidireccional Complejo de *n* bits; (g) Perturbador Bidireccional Simple de *n* bits; (h) Perturbador Bidireccional Complejo de *n* bits

Por último, en [Baraza03] se ha propuesto una redefinición de los perturbadores de la Figura 15, intentando agrupar y generalizar el número de modelos de perturbadores. Esta propuesta se muestra en la Figura 16. Las características de cada perturbador se muestran a continuación:

a) **Perturbador serie unidireccional** (en inglés ***Unidirectional Serial Saboteur***, USS): Es el *perturbador serie simple* de la Figura 15–a, si bien permite inyectar más modelos de fallos que aquél.

b) **Perturbador serie bidireccional** (***Bi-directional Serial Saboteur***, BSS): Es similar al perturbador serie simple bidireccional de la Figura 15–b. Por un lado, se ha eliminado la entrada de control de sentido de inyección (R/W) y, por otro, se ha ampliado el número de modelos de fallo que permite inyectar.

c) **Perturbador serie unidireccional de n bits** (***n-bit Unidirectional Serial Saboteur***, nUSS): Este modelo reemplaza a los tres modelos unidireccionales de la Figura 15.

d) **Perturbador serie bidireccional de n bits** (***n-bit Bi-directional Serial Saboteur***, nBSS): Sustituye a los tres modelos bidireccionales de la Figura 15.

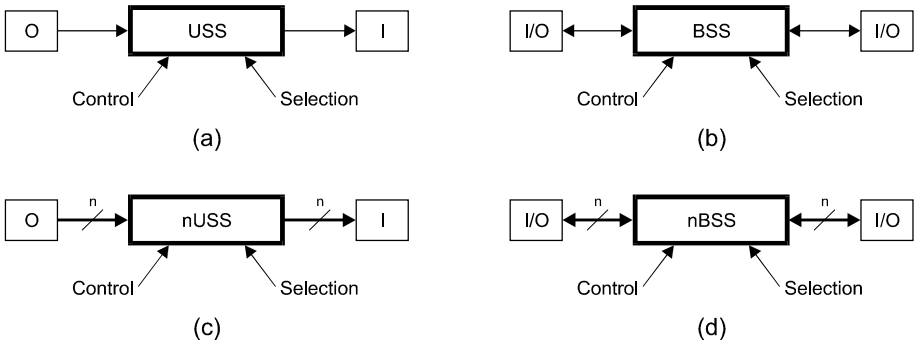


Figura 16. Nuevos perturbadores serie propuestos en [Baraza03]. (a) Perturbador Serie Unidireccional; (b) Perturbador Serie Bidireccional; (c) Perturbador Serie Unidireccional de *n* bits; (d) Perturbador Serie Bidireccional de *n* bits.

A continuación, en los puntos siguientes, se comentan detalladamente las características de todos los perturbadores implementados en esta tesis, cuyo esquema general se puede ver en la Figura 15.

3.2.2.1.1 Perturbador serie simple

Este perturbador interrumpe la conexión entre una salida y su correspondiente receptor o entrada, modificando el valor del receptor. En este caso, el diseño es muy simple. El perturbador tiene una entrada (*I*) y una salida (*O*) más las señales *Control* y *Selección*. En el momento en que se activa la señal *Control*, se inyecta el fallo. La selección del tipo de fallo a inyectar se realiza mediante la señal externa *Selección*. El fallo se inyecta mediante la asignación del valor correspondiente a la señal *O*. La activación de la señal de control determina el instante de inyección y la duración del fallo. Esta señal es controlada en todo momento desde el simulador. En la Figura 17 se puede ver un esquema más detallado de este perturbador, así como del control temporal de la señal de control, donde t_{inj} es el instante de inyección y Δt_{inj} es la duración del fallo.

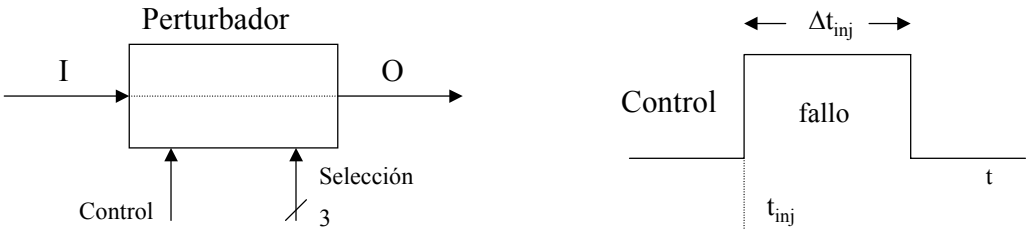


Figura 17. Perturbador Serie Simple y activación del fallo.

A continuación, se muestra un esquema simplificado de la arquitectura comportamental de este perturbador, escrito en pseudo código VHDL:

```
architecture comportamental of perturbador_serie_simple
begin
  process (I, Control)
  begin
    if Control = '1' and not Control'stable then
      selección_tipo_fallo;
      O <= f_inj(I, Selección);
    else
      O <= I;
    end if;
  end process;
end architecture;
```

Como se puede observar, el código del perturbador consiste básicamente en un proceso activado por la señal de entrada y la señal de control. En el ejemplo, el fallo se inyecta sincronizadamente con el flanco de subida de la señal de control (gracias a la condición Control = '1' **and not** Control**'stable**). Obviamente, también es posible sincronizar la inyección del fallo con el flanco de bajada de la señal de control (mediante Control = '0' **and not** Control**'stable**). Un detalle que hay que tener en cuenta es la inyección de fallos permanentes. En este caso, la señal *Control* solo se debe activar una vez y permanecer inalterada durante el resto de la simulación para que el fallo inyectado sea permanente, es decir, la señal de salida no sea reconectada a su *driver*. A nivel práctico, y tal y como se ha comentado anteriormente, la inyección está sincronizada con el flanco de subida de la señal de control. Si queremos inyectar un fallo permanente, solo se debe producir un flanco de subida en dicha señal.

En el pseudo código anterior, se pueden observar dos funciones de interés a la hora de realizar la inyección de fallos. En primer lugar, está la función selección_tipo_fallo. Esta función selecciona el tipo de fallo a inyectar dependiendo del valor de la señal *Selección*. La función f_{inj}(I, Selección) asigna el valor del fallo a la señal de salida dependiendo del tipo de fallo seleccionado. La Tabla 5 muestra los fallos implementados en este perturbador, así como los valores de salida de la función f_{inj} según el tipo de fallo seleccionado [DGil99a].

Tipo de fallo	f_{inj}(I, Selección)
stuck-at '0'	'0'
stuck-at '1'	'1'
*bit-flip***	not(I)
*pulse****	not(I)
*open-line**	'Z' (alta impedancia)
delay	I después del retardo, retardo > 0
indeterminación	'X'
*stuck-open**	'0' después $t_{retención}$

* Exclusivamente fallos permanentes
** En elementos de memoria
*** En lógica combinacional

Tabla 5. Tipos de fallos implementados en el perturbador serie simple.

3.2.2.1.2 Perturbador serie complejo

En este caso, el perturbador interrumpe la conexión entre dos salidas y sus correspondientes receptores, modificando el valor de los receptores. El diseño es algo más complicado que el modelo anterior. El perturbador tiene dos entradas, dos salidas, la señal de control y la señal de selección del tipo de fallo, tal y como se puede ver en la Figura 18. Cuando se activa la señal de control, se inyecta el fallo seleccionado por la señal de selección. Al igual que en el caso anterior, la activación de la señal de control determina el instante de inyección y la duración del fallo y es asimismo controlada por el simulador. A continuación, escrito en pseudo código VHDL, podemos ver el esquema simplificado de la arquitectura comportamental de este perturbador:

```
architecture comportamental of perturbador_serie_complejo
begin
   process (I1, I2, Control)
   begin
      if Control = '1' and not Control'stable then
         selección_tipo_fallo;
         O1 <= f1inj(I1, I2, Selección);
         O2 <= f2inj(I1, I2, Selección);
      else
         O1 <= I1;
         O2 <= I2;
      end if;
   end process;
end architecture;
```

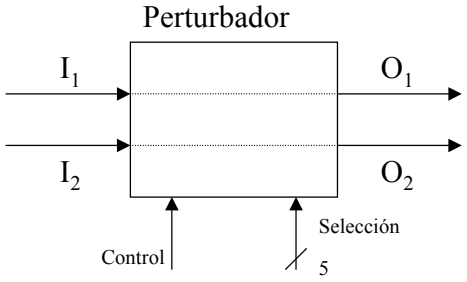


Figura 18. Perturbador Serie Complejo.

Como se puede ver en el pseudo código anterior, el funcionamiento de este perturbador es similar al funcionamiento del perturbador anterior. Se tiene un proceso que inyecta el fallo al producirse un evento en la señal *Control* (flanco de subida en el pseudo código del ejemplo). En este momento se selecciona el tipo de fallo para cada salida en función de las entradas y de la señal de selección. En este caso, existe un mayor número potencial de modelos de fallos inyectables que en el perturbador serie simple. La Tabla 6 muestra el valor de $f1_{inj}$(I1, I2, Selección) y de $f2_{inj}$(I1, I2, Selección) según el fallo a inyectar [DGil99a]. La señal de selección del fallo también es controlada por el simulador.

Tipo de fallo	**$f1_{inj}$(I1, I2, Selección)**	**$f2_{inj}$(I1, I2, Selección)**
stuck-at '0' en O1	'0'	I2
stuck-at '1' en O1	'1'	I2
*bit-flip*** en O1	not(I1)	I2
*pulse**** en O1	not(I1)	I2
*open-line** en O1	'Z' (alta impedancia)	I2
delay en O1	I1 después del retardo, retardo > 0	I2
indeterminación en O1	'X'	I2
stuck-at '0' en O2	I1	'0'
stuck-at '1' en O2	I1	'1'
*bit-flip*** en O2	I1	not(I2)
*pulse**** en O2	I1	not(I2)
*open-line** en O2	I1	'Z' (alta impedancia)
delay en O2	I1	I2 después del retardo, retardo > 0
indeterminación en O2	I1	'X'
*stuck-open** en O1	'0' después $t_{retención}$	I2
*stuck-open** en O2	I1	'0' después $t_{retención}$

* Exclusivamente fallos permanentes
** En elementos de memoria
*** En lógica combinacional

Tabla 6. Tipos de fallos implementados en el perturbador serie complejo.

3.2.2.1.3 Perturbador bidireccional serie simple

Este perturbador tiene una entrada y una salida de datos (*I/O*) bidireccionales, una entrada de control (*Control*) y de selección (*Selección*) más una entrada de lectura/escritura (*L/E*). Dependiendo del valor de la señal *L/E*, se modifica la entrada o la salida, es decir, esta señal determina la dirección de la transferencia de información. Como en los perturbadores anteriores, la señal de control activa la inyección, siendo controlada desde el simulador. La activación de esta señal determina el instante de inyección y la duración del fallo. La selección del tipo de fallo se realiza mediante la señal externa *Selección*, también controlada por el simulador. La Figura 19 muestra un esquema más detallado de este perturbador.

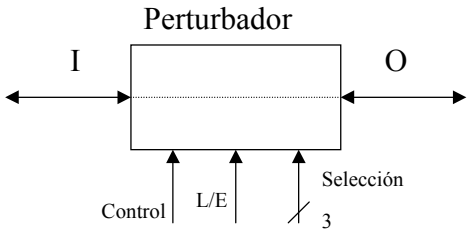


Figura 19. Perturbador Bidireccional Serie Simple.

La arquitectura comportamental de este perturbador consiste básicamente en un proceso activado por las señales *I*, *O*, *L/E* y *Control*. Si se activa la señal *Control*, se inyecta el fallo según la selección llevada a cabo por la señal *Selección*. La perturbación puede afectar a las señales *O* o *I*, dependiendo del valor de la señal *L/E*. Si la señal *Control* no se activa, se actualiza *O(I)* con el nuevo valor de *I(O)*, según el funcionamiento normal (sin fallos) del sistema. A continuación, se muestra un esquema simplificado de esta arquitectura, escrito en pseudo código VHDL:

```
architecture comportamental of perturbador_bidir_serie_simple
begin
  process (I, O, L/E, Control)
  begin
    if Control = '1' and not Control'stable then
      selección_tipo_fallo;
      if L/E ='1' then
        O <= finj(I, Selección);
      else
        I <= finj(O, Selección);
      end if;
    else
      if L/E ='1' then
        O <= I;
      else
        I <= O;
      end if;
    end if;
  end process;
end architecture;
```

La Tabla 7 muestra los modelos de fallos implementados por este perturbador [DGil99a]. Como se puede observar, la principal diferencia con los modelos de fallos del perturbador serie simple es la dirección de la información, que tal y como se ha comentado anteriormente, depende de la señal de L/E. Una posible mejora de este perturbador se presenta en [Baraza03], donde se elimina la señal de L/E simplificando el perturbador, y por lo tanto, disminuyendo el tamaño de las trazas de simulación. Para realizar esta eliminación, lo que se hace es controlar en cada momento cuál ha sido el último puerto modificado (que puede ser *I* o *O*), el cual indicará el flujo de información, y de ese modo, dónde inyectar.

Tipo de fallo	f_{inj}(entrada**, Selección)
stuck-at '0'	'0'
stuck-at '1'	'1'
*bit-flip****	not(I)
*pulse*****	not(I)
*open-line**	'Z' (alta impedancia)
delay	I después del retardo, retardo > 0
indeterminación	'X'
*stuck-open**	'0' después $t_{retención}$

* Exclusivamente fallos permanentes
** I o O, dependiendo del valor de L/E
*** En elementos de memoria
**** En lógica combinacional

Tabla 7. Tipos de fallos implementados en el perturbador bidireccional serie simple.

3.2.2.1.4 Perturbador bidireccional serie complejo

Este perturbador tiene dos entradas y dos salidas de datos (I_n/O_n), una entrada de control (*Control*) y otra para la selección del fallo a inyectar (*Selección*) más una entrada *L/E*, que indica la dirección del flujo de información. La Figura 20 muestra un esquema detallado de este perturbador.

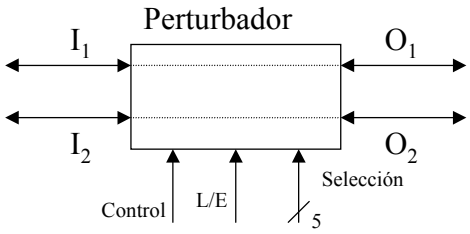


Figura 20. Perturbador Bidireccional Serie Complejo.

El funcionamiento de este perturbador es muy similar al funcionamiento del perturbador anterior: cuando se produce un evento en la señal de control (un flanco), se activa la inyección. Este evento determina el instante de inyección así como la duración del fallo. El tipo de fallo a inyectar se selecciona en función de la señal *Selección* y de las señales de entrada y de salida. Dependiendo del valor de la señal *L/E*, el valor modificado será inyectado en las señales de salida o de entrada. A continuación, podemos ver el esquema simplificado de la arquitectura comportamental de este perturbador, escrito en pseudo código VHDL:

```
architecture comportamental of perturbador_bidir_serie_complejo
begin
  process (I1, I2, O1, O2, L/E, Control)
  begin
    if Control = '1' and not Control'stable then
      selección_tipo_fallo;
      if L/E ='1' then
        O1 <= f1inj(I1, I2, Selección);
        O2 <= f2inj(I1, I2, Selección);
      else
        I1 <= f1inj(O1, O2, Selección);
        I2 <= f2inj(O1, O2, Selección);
```

```
        end if;
      else
        if L/E ='1' then
          O1 <= I1;
          O2 <= I2;
        else
          I1 <= O1;
          I2 <= O2;
        end if;
      end if;
    end process;
  end architecture;
```

Este perturbador permite inyectar más fallos que el anterior. En particular, los nuevos modelos de fallos que aporta este perturbador son el cortocircuito en las líneas de salida y líneas cruzadas, tal y como se puede ver en la Tabla 8 [DGil99a].

Tipo de fallo	**$f1_{inj}$(I1, I2, Selección)**	**$f2_{inj}$(I1, I2, Selección)**
stuck-at '0' en O1	'0'	I2
stuck-at '1' en O1	'1'	I2
*bit-flip*** en O1	not(I1)	I2
*pulse**** en O1	not(I1)	I2
*open-line** en O1	'Z' (alta impedancia)	I2
delay en O1	I1 después del retardo, retardo > 0	I2
indeterminación en O1	'X'	I2
stuck-at '0' en O2	I1	'0'
stuck-at '1' en O2	I1	'1'
*bit-flip*** en O2	I1	not(I2)
*pulse**** en O2	I1	not(I2)
*open-line** en O2	I1	'Z' (alta impedancia)
delay en O2	I1	I2 después del retardo, retardo > 0
indeterminación en O2	I1	'X'
cortocircuito (*short*) en las salidas	I1	I1
	I2	I2
líneas cruzadas	I2	I1
*stuck-open** en O1	'0' después $t_{retención}$	I2
*stuck-open** en O2	I1	'0' después $t_{retención}$

* Exclusivamente fallos permanentes
** En elementos de memoria
*** En lógica combinacional

Tabla 8. Tipos de fallos implementados en el perturbador bidireccional serie complejo.

Una mejora de este perturbador sería el añadir una segunda señal de dirección *L/E*. Con esta señal extra se conseguirían tener direcciones de inyección diferentes para las parejas de señales (*I1-O1*) e (*I2-O2*). Es decir, con la primera señal *L/E*, se inyectarían fallos en las señales (*I1-O1*), mientras que con la segunda señal *L/E*, los fallos se inyectarían en la pareja (*I2-O2*).

3.2.2.1.5 Perturbador unidireccional simple de n *bits*

Este perturbador se utiliza para *buses* unidireccionales (por ejemplo, *buses* de direcciones y de control). Se forma con la unión de *n* perturbadores serie simple, tal y como se puede ver en la Figura 21. Este perturbador puede interrumpir la conexión entre una o más salidas y sus correspondientes receptores o entradas, modificando el valor del receptor cuando el fallo es activado. Como en los perturbadores anteriores, en el momento en que se activa la señal

Control$_i$, se establece el instante de inyección del fallo y su duración. El tipo de fallo a inyectar se selecciona en función del valor de la señal *Selección*, asignándose el valor correspondiente a la señal O_i. El funcionamiento de este perturbador es una generalización del funcionamiento del perturbador serie simple. Cabe destacar el hecho de la utilización de una única señal de selección del tipo de fallo, con el ahorro consiguiente de señales. El perturbador activo se distinguirá de los demás por la activación de la señal de control correspondiente [DGil99a].

Este perturbador también se podría utilizar para la inyección de fallos múltiples. Esto se lograría con la activación de más de una señal *Control*. Si además se quiere inyectar más de un modelo de fallo (durante la inyección de fallos múltiples), se debería tener una señal *Selección* para cada perturbador serie simple, aunque lo más representativo en *buses* es que fallen las salidas en grupo con el mismo tipo de fallo, por ejemplo, *stuck-at* '1' o *stuck-at* '0' múltiples, correspondientes a fallos simultáneos permanentes en los *drivers*.

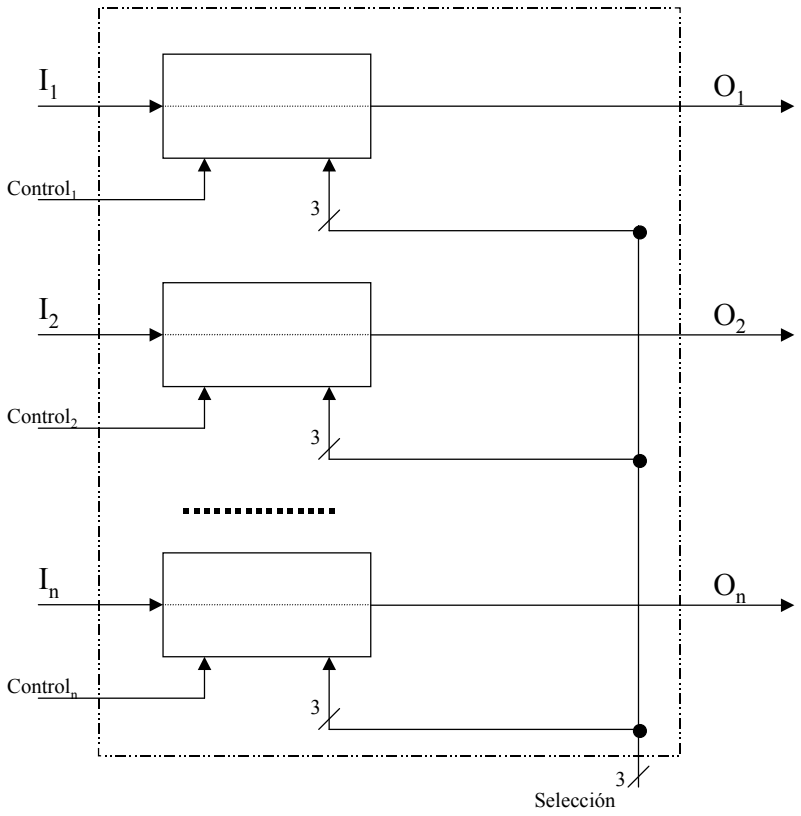


Figura 21. Perturbador Unidireccional Simple de *n* bits.

En este caso, el pseudo código de la arquitectura comportamental de este perturbador es simplemente una sentencia generate, la cual instancia el número de perturbadores adecuado para el tamaño del *bus* al que va a modificar. Este pseudo código se puede ver a continuación.

```
architecture comportamental of perturbador_unidir_serie_simple_n_bits
begin
  for i in n'range generate
   c_p : perturbador_serie_simple generic map (…)
                                  port map (I => entrada_perturb_unidir(i),
                                            O => salida_perturb_unidir (i),
                                            Control => Control(i),
                                            Selección => Selección);
  end generate;
```

end architecture;

Al ser este modelo una generalización del perturbador serie simple, los fallos inyectados son los mismos de éste. Estos modelos de fallos que pueden ser inyectados se muestran en la Tabla 5.

3.2.2.1.6 Perturbador unidireccional complejo de n *bits*

Al igual que el caso anterior, este perturbador se utiliza con *buses* unidireccionales (por ejemplo, *buses* de direcciones y de control) de *n* bits. El diseño de este perturbador presenta dos características específicas:

i. Se han utilizado perturbadores serie complejos (para un *bus* de *n* bits, se han utilizado *n/2* perturbadores).

ii. Debido a los perturbadores base utilizados, el número de fallos que se puede inyectar es mayor en este perturbador que en el anterior.

La Figura 22 muestra un esquema detallado de la estructura de este perturbador, el cual está compuesto por *n* señales de entrada, *n* señales de salida, la señal de selección del tipo de fallo a inyectar y *n/2* señales de control. El funcionamiento de este perturbador es similar al funcionamiento del perturbador serie complejo. El perturbador interrumpe la conexión entre dos salidas y sus correspondientes receptores, modificando el valor de los mismos. El fallo se inyecta al activarse la señal de control correspondiente, determinando el instante de inyección y la duración del fallo. Como en el caso anterior, la arquitectura de este perturbador se compone de una sentencia generate, la cual instancia el número de perturbadores adecuado para el tamaño del *bus*. Cabe destacar que los perturbadores base tienen dos entradas y dos salidas, con lo que se necesitan *n/2* perturbadores para un *bus* de *n* bits, mientras que el número de señales de control es también de *n/2* [DGil99a].

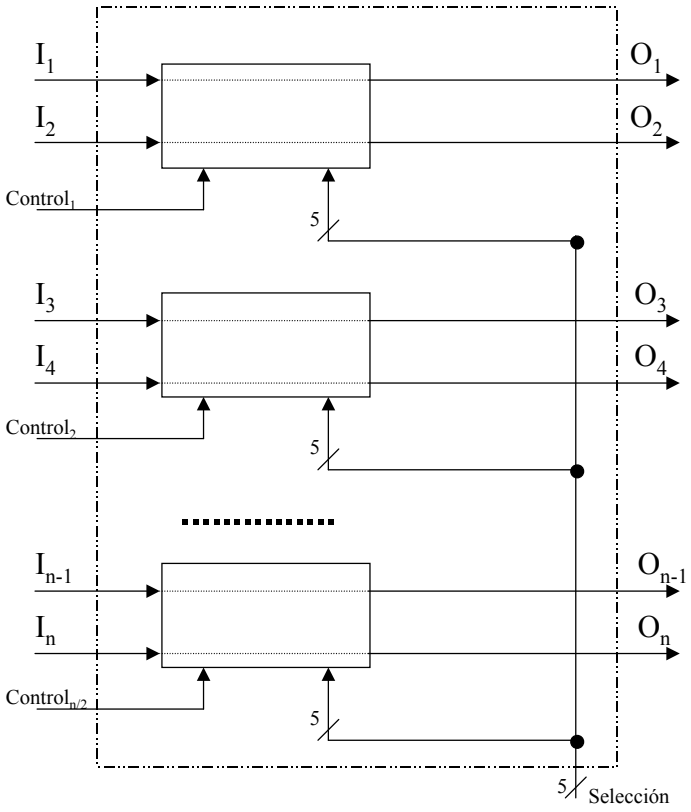


Figura 22. Perturbador Unidireccional Complejo de *n* bits.

A continuación se puede ver un esquema de la arquitectura comportamental de este perturbador escrito en pseudo código VHDL. Al ser este perturbador una generalización del perturbador serie complejo, los modelos de fallos que puede inyectar son los mismos de este perturbador, los cuales se muestran en la Tabla 6.

```
architecture comportamental of perturbador_unidir_complejo_n_bits
begin
  for i in (n/2)'range generate
   c_p : perturbador_serie_simple generic map (…)
                                  port map (I1 => entrada_perturb_bidir(i*2),
                                            I2 => entrada_perturb_bidir(i*2 + 1),
                                            O1 => salida_perturb_bidir(i*2),
                                            O2 => salida_perturb_bidirO(i*2 + 1),
                                            Control => Control(i),
                                            Selección => Selección);
  end generate;
end architecture;
```

Al igual que en el perturbador unidireccional simple de *n* bits, el perturbador unidireccional complejo de *n* bits también se podría utilizar para la inyección de fallos múltiples, activando para ello más de una señal *Control*. Para inyectar más de un modelo de fallo, se debería tener una señal *Selección* para cada perturbador serie simple.

3.2.2.1.7 Perturbador bidireccional simple de n *bits*

Los siguientes perturbadores se han diseñado como complemento de los dos perturbadores anteriores. En este caso, el perturbador bidireccional simple de *n* bits se utiliza para *buses* bidireccionales (por ejemplo, *buses* de datos y de control) de *n* bits. Su diseño se ha realizado mediante la unión de *n* perturbadores bidireccionales serie simple. La Figura 23 muestra un esquema más detallado de este perturbador. El perturbador se compone de *n* señales de entrada, *n* señales de salida, *n* señales de control, una señal de selección del tipo de fallo a inyectar y una señal de dirección *L/E*. El funcionamiento es similar al funcionamiento de los perturbadores anteriores. Cuando se activa la señal de control (se produce un flanco en esta señal), se inyecta el fallo. Esta activación determina tanto la duración del fallo como el instante de inyección del mismo. El tipo de fallo se selecciona en función del valor de la entrada y la salida correspondiente así como del valor de la señal de selección, que es única para todo el modelo. El fallo se inyecta en la entrada o en la salida en función del valor de la señal *L/E*.

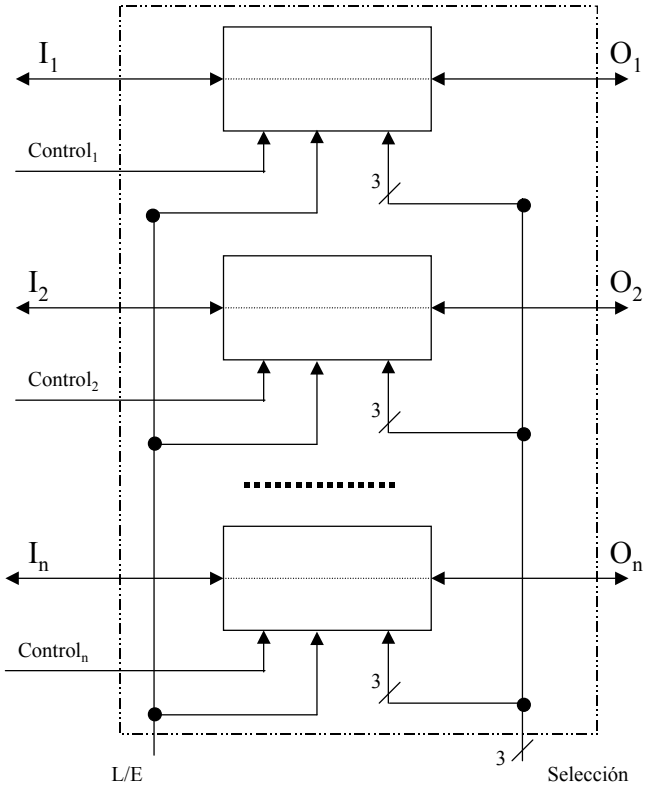


Figura 23. Perturbador Bidireccional Simple de *n* bits.

Como se puede ver en el esquema de la arquitectura comportamental de este perturbador en pseudo código VHDL, para implementar este perturbador se ha utilizado una sentencia generate, la cual establece el número de perturbadores adecuado para el tamaño del *bus* al que va a afectar. Al ser este perturbador una generalización del perturbador bidireccional serie simple, se pueden inyectar los mismos fallos de éste, los cuales se pueden ver en la Tabla 7. Al igual que en los dos perturbadores anteriores, con este perturbador también se podrían inyectar fallos múltiples y más de un modelo de fallos. Además, con el diseño presentado en [Baraza03], comentado en el punto 3.2.2.1.3, se podrían eliminar las señales de L/E con el fin de plantear un diseño más óptimo.

```
architecture comportamental of perturbador_bidir_simple_n_bits
begin
   for i in n'range generate
    c_p : perturbador_bidir_serie_simple generic map (…)
                                  port map (I => entrada_perturb_bidir_simple(i),
                                            O => salida_perturb_bidir_simple(i),
                                            Control => Control(i),
                                            Selección => Selección,
                                             L/E => L/E);
   end generate;
end architecture;
```

3.2.2.1.8 Perturbador bidireccional complejo de n *bits*

El perturbador bidireccional complejo de *n* bits se utiliza para *buses* bidireccionales (por ejemplo, *buses* de datos y de control), pero en este caso, su diseño se ha realizado mediante la unión de *n/2* perturbadores bidireccionales serie complejos. La Figura 24 muestra un esquema más detallado de este perturbador.

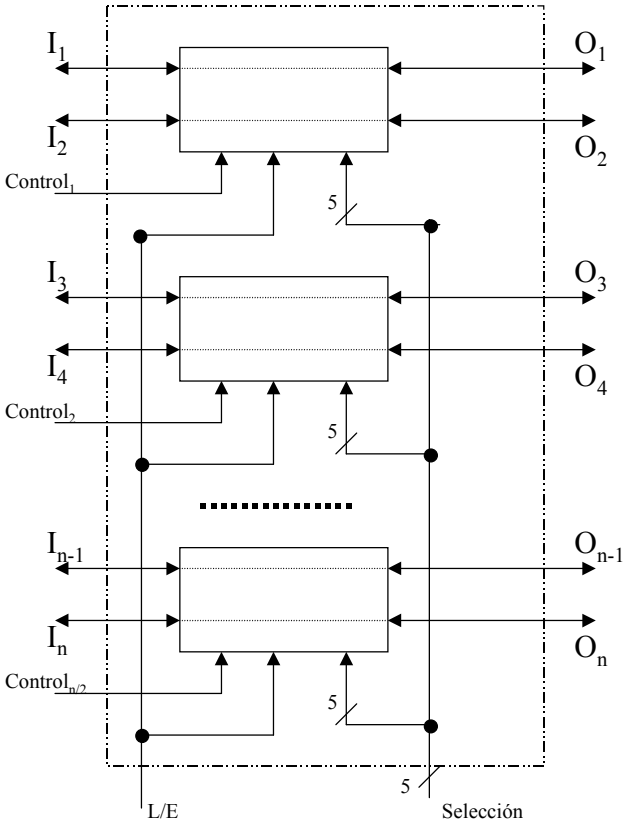


Figura 24. Perturbador Bidireccional Complejo de *n* bits.

El perturbador se compone de *n* señales de entrada, *n* señales de salida, *n/2* señales de control, una señal de selección del tipo de fallo a inyectar y una señal de dirección *L/E*. El funcionamiento es similar al funcionamiento de los perturbadores anteriores. Cuando se activa la señal de control, se inyecta el fallo. Esta activación determina tanto la duración del fallo como el instante de inyección del mismo. El tipo de fallo se selecciona en función del valor de la entrada y la salida correspondiente así como del valor de la señal de selección, que es única para todo el modelo. Cabe destacar el ahorro en el número de señales de control necesarias, debido a la estructura de los perturbadores base utilizados. Al igual que pasa con el perturbador básico (perturbador bidireccional serie complejo), el fallo se inyectará en la señal de entrada o de salida dependiendo del valor de la señal *L/E*. Al igual que en el perturbador bidireccional serie complejo, se podrían añadir más señales de dirección *L/E* al perturbador bidireccional complejo de *n* bits, con lo que se aumentarían las combinaciones en las señales (*I-O*), posibilitando la inyección de un número mayor de modelos de fallos. Además, también se podrían inyectar fallos múltiples mediante la activación de las señales de *Control* necesarias, y para inyectar más de un modelo de fallo, se debería tener una señal *Selección* para cada perturbador bidireccional serie complejo, tal y como se ha comentado anteriormente.

Al igual que los perturbadores anteriores, y tal y como se puede ver en el esquema de la arquitectura comportamental de este perturbador en pseudo código VHDL, para implementar este perturbador se ha utilizado una sentencia generate, la cual establece el número de perturbadores adecuado para el tamaño del *bus* al que va a afectar. Al ser este perturbador una

generalización del perturbador bidireccional serie complejo, se pueden inyectar los mismos fallos de éste, los cuales se pueden ver en la Tabla 8.

```
architecture comportamental of perturbador_bidir_complejo_n_bits
begin
  for i in (n/2)'range generate
   c_p : perturbador_bidir_serie_complejo generic map (…)
                        port map (I1 => entrada_perturb_bidir_complejo(i*2),
                                  I2 => entrada_perturb_bidir_complejo(i*2 + 1),
                                  O1 => salida_perturb_bidir_complejo(i*2),
                                  O2 => salida_perturb_bidir_complejo(i*2 + 1),
                                  Control => Control(i),
                                  Selección => Selección,
                                  L/E => L/E);
  end generate;
end architecture;
```

3.2.2.1.9 Modelos de fallos en perturbadores

Una de las ventajas de los perturbadores respecto a la técnica de las órdenes del simulador es la ampliación del modelo de fallos que es posible inyectar. Respecto a los modelos de fallo mencionados en punto 3.2.1.3, la técnica de los perturbadores permite inyectar los modelos de indeterminación, *delay*, *bit-flip* (en celdas de memoria o registros), *pulse* (en lógica combinacional). Además del modelo *stuck-at* ('0', '1'), se amplían los modelos de fallos permanentes con los modelos *open-line*, *stuck-open*, *short* y *bridging*. En el punto 4.3, "Modelos de fallos", se profundizará en el significado y los mecanismos físicos de estos fallos.

3.2.2.2 Mutantes

Se puede definir a los mutantes como nuevos componentes VHDL que reemplazan a los componentes originales. Durante el funcionamiento normal, el componente mutado se comporta como el componente al que reemplaza. Sin embargo, cuando se inyecta el fallo, el componente mutado simula el comportamiento del componente original, pero en presencia de fallos. La mutación puede realizarse de distintas maneras [Jenn94b, Rimén97, Baraza99, Gracia01a, Gracia01b, DGil03b]:

- Añadiendo perturbadores a descripciones estructurales o comportamentales de los componentes.
- Modificando descripciones estructurales mediante el reemplazo de componentes, por ejemplo, una puerta NAND puede ser reemplazada por una puerta NOR.
- Modificando estructuras sintácticas de descripciones comportamentales.

En un principio, para un modelo específico escrito en VHDL, existe un número casi infinito de mutaciones, aunque se puede considerar un subconjunto representativo de mutaciones en el nivel lógico. [Armstrong92] menciona ocho modelos de fallos. Estos modelos pueden ser clasificados en dos grupos (control o datos), dependiendo del flujo que perturben:

- Fallos en el flujo de control: *stuck-then*, *stuck-else*, *assignment control*, *dead process*, *dead clause* y *micro-operations*.
- Fallos en el flujo de datos: *local stuck-data* y *global stuck-data*.

La Tabla 9 muestra estos ocho modelos de fallos implementados mediante la modificación de unidades sintácticas en descripciones comportamentales.

Nombre del fallo	**Modificación efectuada**
stuck-then	Cambio de la condición por el valor cierto (*true*).
stuck-else	Cambio de la condición por el valor falso (*false*).
assignment control	Perturbación de una operación de asignación.
dead process	Eliminación de la lista de sensibilización de un proceso.
dead clause	Eliminación de una condición en una sentencia *case*.
micro-operation	Perturbación de un operador.
local stuck-data	Perturbación del valor de una variable, constante o señal en una expresión.
global stuck-data	Eliminación de todas las modificaciones del valor de una variable o una señal en una arquitectura.

Tabla 9. Modelos de fallos implementados con mutantes [Armstrong92].

Anteriormente, en diferentes trabajos se han comentado otros modelos algorítmicos para fallos en el flujo de control, como por ejemplo en [Ghosh91]. Algunos de ellos coinciden con los especificados en la Tabla 9, mientras que otros incluyen modificaciones en la sincronización y en las cláusulas temporales (cláusulas *after* y *wait*).

Una propuesta posterior se presenta en [Riesgo96], orientando la clasificación de los modelos de fallos al modelado de fallos *hardware* en descripciones sintetizables en VHDL. Sin embargo, dicha clasificación parte de una serie de supuestos:

- Sólo se producen fallos simples y todos los fallos son permanentes.
- Sólo es válida para modelos comportamentales:
  - ⇨ Para modelos estructurales asume como referencia el modelo de fallo *stuck-at* (‘0’ y ‘1’) en las líneas de interconexión entre componentes.
  - ⇨ No se permiten modelos mixtos (parte estructural y parte comportamental).

A partir de estas suposiciones, se propone una clasificación que distingue tres clases de fallos: fallos en datos, fallos en expresiones y fallos en sentencias. Bastantes de ellos coinciden con los expuestos en la Tabla 9.

Más recientemente, en [Leveugle00b], se propone un nuevo modelo de fallo para descripciones comportamentales de máquinas de estados denominado “*transición errónea*”. Este fallo consiste en forzar a la máquina de estados a realizar una transición distinta de la que debería hacer en un momento determinado. Este modelo de fallo estaría cubierto por uno de los modelos de fallos de [Riesgo96], ya que en VHDL las máquinas de estados se implementan fácilmente como datos de un tipo enumerado que contiene los nombres de los estados, y este caso general está incluido en dicha clasificación.

Una de las opciones para poder implementar los mutantes es el uso del mecanismo de configuraciones (en inglés *configuration*) del VHDL. Esta unión arquitectura–componente es estática [DGil98b, DGil99a, Gracia01b], es decir, una vez que se ha compilado una configuración específica, ésta no puede ser cambiada durante la simulación. Por esta razón, utilizando este mecanismo sólo se pueden inyectar fallos permanentes. La implementación de fallos transitorios mediante la técnica de los mutantes requiere la realización de instanciaciones dinámicas. En [Jenn92] se sugiere un método para conseguir el cambio dinámico de arquitecturas durante la simulación, que se podría utilizar para nuestros objetivos.

Este método está basado en el uso de “*asignaciones con guardas*” en bloques (en inglés *guarded assignments*), así como el mecanismo de configuración. Las “*asignaciones con guardas*” permiten la activación dinámica de bloques y las arquitecturas asociadas a los mismos. Una “*asignación con guarda*” [Perry94] consiste en una expresión de asignación en una señal, condicionada por una expresión *booleana* (expresión de guarda). Si la expresión de guarda es cierta, se ejecuta la asignación. Si no, se genera una asignación nula. La Figura 25

muestra un esquema de este método para ejecutar el cambio dinámico entre la arquitectura sin fallos y una arquitectura mutada del sistema mediante el uso de asignaciones con guardas en bloques.

El funcionamiento es el siguiente. Si *elec* = 1, se habilita el bloque 1. Este bloque está asociado a la configuración 1, que en este caso se corresponde con la configuración sin fallos. Es decir, si *elec* = 1, se habilita el bloque 1, el cual asigna la arquitectura sin fallo a la entidad global del sistema. En cambio, si *elec* = 2, es el bloque 2 el que se habilita. Este bloque está asociado a la configuración 2, que en este caso se corresponde con la arquitectura mutada, asignando de esta manera la arquitectura mutada a la entidad global del sistema, inyectándose de esta manera el fallo [Gracia01a, Gracia01b].

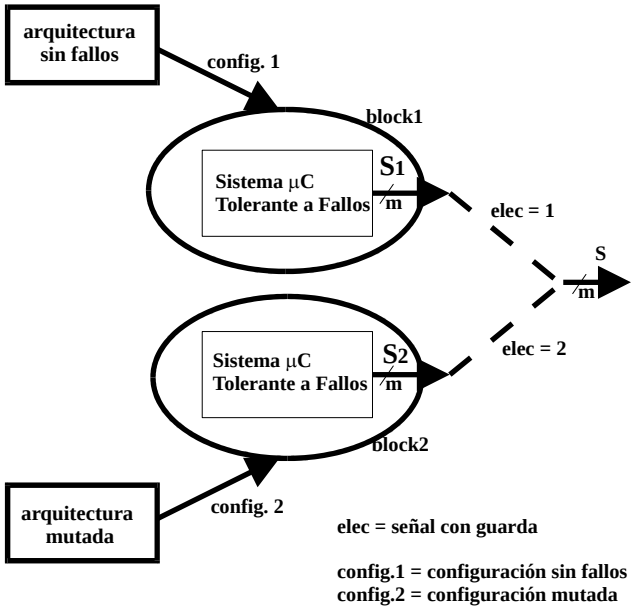


Figura 25. Implementación de mutantes transitorios. Modificación dinámica de la arquitectura [Gracia01b].

La activación dinámica de bloques implica también la multiplexación de señales de la arquitectura estructural del sistema. En la Figura 25 podemos ver *S1* y *S2*. Estas señales se asignan a la señal general *S* según el valor de la señal con guarda.

La Figura 26 muestra las arquitecturas sin fallos y mutadas. Se ha considerado el caso general de una arquitectura estructural con componentes que pueden ser estructurales o comportamentales.

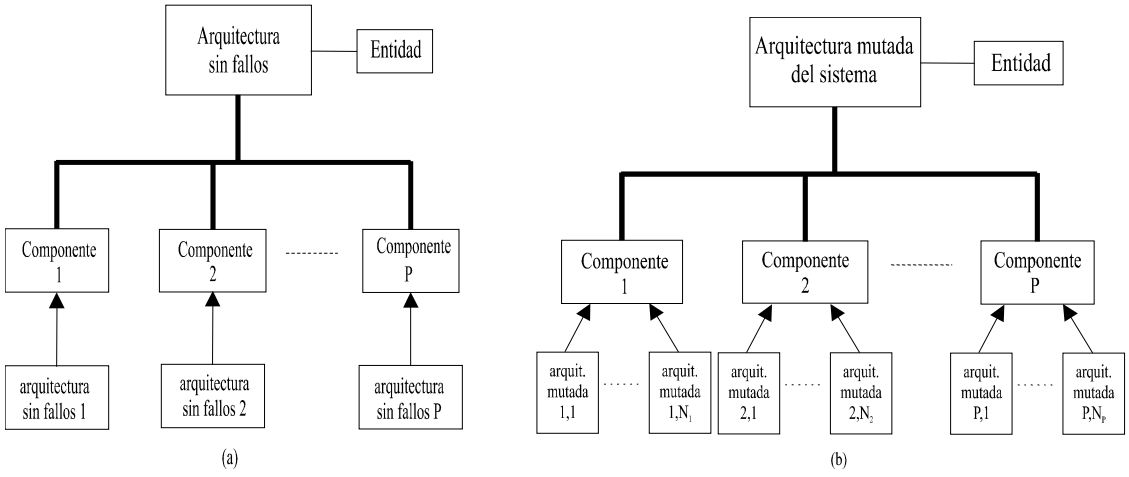


Figura 26. (a) Arquitectura sin fallos; (b) Arquitectura mutada.

Una vez creadas las réplicas de los componentes originales, la inyección se realiza al elegir una serie de combinaciones de todas las arquitecturas disponibles a través de la sentencia *configuration* del VHDL y simular esta combinación específica. En principio existen N_i arquitecturas mutadas diferentes del componente *i*. Aunque todas las arquitecturas están asignadas al mismo componente, solamente una de ellas es seleccionada en un momento dado. La unión componente–arquitectura mutada se especifica en la declaración de la arquitectura mutada [DGil99a, Gracia01a].

Para obtener los distintos mutantes ((i,1), (i,2),…, (i,N_i)), se han modificado las estructuras sintácticas de las descripciones comportamentales [Ghosh91, Armstrong92]. Si dentro de las arquitecturas existen varios procesos, se selecciona uno de ellos aleatoriamente. Se han considerado fallos individuales, es decir, en una configuración mutada solamente se especifica un mutante de un componente, y dicho mutante incluye un modelo de fallo simple. A continuación se puede ver la secuencia de pseudo órdenes que implementa la activación dinámica de arquitecturas que se muestra en la Figura 25 [Gracia01a, Gracia01b]:

1. Señal de guarda <= 1
2. Simular hasta el instante de inyección
3. Almacenar el estado
4. Escribir el estado en la arquitectura mutada
5. Señal de guarda <= 2
6. Simular durante la duración del fallo
7. Guardar el estado
8. Escribir el estado en la arquitectura sin fallos
9. Simular hasta el final de la simulación
10. Escribir el fichero de traza

Estas pseudo órdenes se pueden basar en algunas órdenes del simulador, facilitándose la automatización de esta técnica, como se comentará posteriormente. El pseudo código anterior muestra varios pasos a tener en cuenta. En primer lugar, se tiene el almacenamiento del estado (paso 3). Esta acción consiste en salvar los valores de las señales y variables de la arquitectura sin fallos en un fichero intermedio. Este fichero se utilizará en el paso 4 con el fin de recuperar los valores de las señales y variables de la arquitectura sin fallos y asignar dichos valores a las señales y variables de la arquitectura mutada. Los siguientes puntos de atención son los pasos 7 y 8. En este caso, el paso 7 almacena el valor de las señales y variables de la arquitectura mutada en el fichero intermedio mientras que el paso 8 graba los valores de las señales y variables, salvados en el fichero intermedio, en la arquitectura sin fallos. Todo este trasiego de datos es debido a características intrínsecas del simulador utilizado durante los experimentos de inyección [Model98, Model01a], como se explica a continuación.

A pesar de que la idea de los bloques con guardas es sintácticamente correcta y funciona, el problema que surge durante la simulación es que las dos arquitecturas (la arquitectura sin fallos y la arquitectura mutada) deben estar activas simultáneamente, es decir, a pesar de que se puede realizar el cambio dinámicamente, la arquitectura mutada se está simulando desde el principio del experimento, lo que nos llevaría a tener un fallo permanente desde el principio de la simulación. Para solucionar este problema y poder inyectar fallos transitorios, se realiza el paso intermedio de almacenar los valores de las señales y variables a través de un fichero.

Para inyectar fallos permanentes, a partir de un instante dado, se deben omitir los pasos 6, 7 y 8 del pseudo código anterior. La Figura 27 muestra el diagrama temporal de la secuencia de pseudo órdenes utilizadas durante la simulación. Hay que detallar que T_{inj} es el instante de inyección del fallo, ΔT_{inj} es el valor de la duración del fallo, T_{simul} es el tiempo de simulación y

elec es la señal con guarda que nos permite el cambio de arquitectura, o lo que es lo mismo, la inyección del fallo.

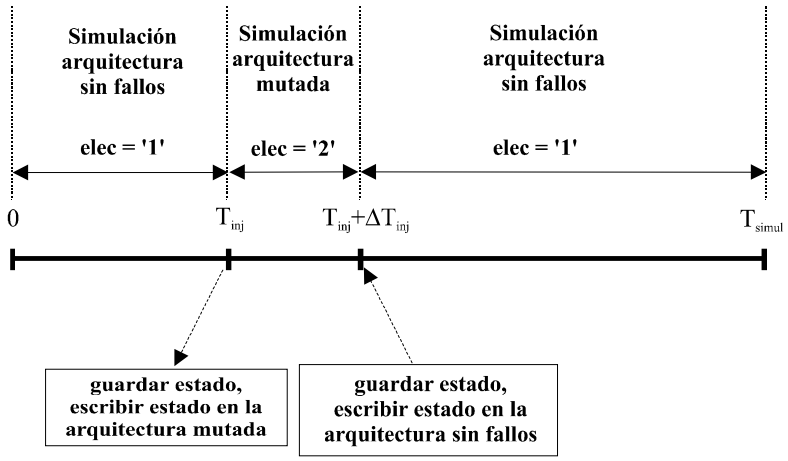


Figura 27. Implementación de mutantes transitorios. Diagrama temporal [Gracia01b].

Sin embargo, esta implementación presenta el problema del coste temporal. El cambio de estado de una arquitectura a otra (*no mutada–mutada* y *mutada–no mutada*) ralentiza de tal manera la simulación que convierte a esta técnica en algo poco práctico. Aunque en el capítulo siguiente se comentará el coste (temporal y espacial) de las tres técnicas de inyección de fallos, en este punto vamos a comentar una solución a este problema propuesta en [Baraza03]. Aunque el método presentado no es demasiado original, sin embargo permite salvar el problema temporal y realizar las inyecciones con un coste temporal despreciable. Básicamente, la idea consiste en incluir todos los mutantes dentro de una única arquitectura general, con el fin de eliminar el proceso de cambio de estado y así, reducir sustancialmente el coste temporal del mismo. Esta arquitectura nueva contendría todos los mutantes del componente (según la Tabla 9 o cualquier otra selección de mutantes que se haya hecho). La mutación se elegiría mediante sentencias if o case, lo que conlleva que a la arquitectura original se le añadirían un cierto número de señales de control, de manera similar al proceso realizado con los perturbadores. Un ejemplo se puede ver en la Tabla 10. En la columna (a) podemos ver un ejemplo simple donde se realiza una suma. En la columna (b) se pueden ver dos mutaciones y el método para activarlas. Como se ha comentado, se ha añadido una señal de control, que activará una de las mutaciones en función de su valor.

```
architecture ejemplo1 of mutacion is
begin
  a <= b + c;
end ejemplo1;
```

(a)

```
architecture ejemplo1_mutado of mutacion is
begin
   process (b, c, control1)
      if (control1 = 0) then a <= b + c;
      elsif (control1 = 1) then a <= b - c;    -- Mutación 1
      elsif (control1 = 2) then a <= b + b; -- Mutación 2
      end if;
  end process;
end ejemplo1;
```

(b)

Tabla 10. (a) Ejemplo de código VHDL. (b) Adición de mutantes según [Baraza03].

El problema obvio que conlleva este método es el aumento de la complejidad espacial de cada arquitectura mutada, ya que cada una de estas arquitecturas debe incluir todos los mutantes de la misma, pudiendo provocar un aumento considerable en el tamaño del fichero correspondiente. Sin embargo, este problema se compensa con las bondades del método, como son la eliminación del cambio de estado de la arquitectura mutada a la arquitectura no mutada, ahorrando una gran cantidad de tiempo, tal y como se verá en el capítulo siguiente. También hay

que resaltar la facilidad de la inyección, ya que se realizará exactamente igual que en los perturbadores, con órdenes del simulador para activar y desactivar las señales de control que determinan la inyección de un fallo.

3.3 Automatización de las técnicas de inyección en modelos en VHDL

Una vez que se han definido tanto las técnicas de inyección de fallos como los modelos de fallos correspondientes, para poder utilizarlos de forma efectiva, se hace necesaria su automatización. En este apartado se va a explicar cómo se ha realizado esta automatización, dejando para el capítulo siguiente la explicación de los experimentos realizados con las diferentes técnicas de inyección.

3.3.1 Automatización de las órdenes del simulador

Las dos secuencias de inyección de fallos mostradas en las secciones 3.2.1.1 y 3.2.1.2 se pueden automatizar fácilmente mediante su inclusión en un fichero de *macro*, posibilidad permitida por el simulador utilizado durante los experimentos de inyección [Model98, Model01a]. Podemos definir un fichero de *macro* como un pequeño programa interpretable por el simulador que permite la ejecución de ciertas órdenes del mismo. En este caso, el pseudo código con la secuencia de órdenes a utilizar para la inyección de fallos en señales y variables se puede ver en la Tabla 11:

Inyección en señales	Inyección en variables
1. **Simular Hasta** *[instante inyección]*	1. **Simular Hasta** *[instante inyección]*
2. **Modificar Señal** *[nombre señal] [valor del fallo]*	2. **Asignar Valor Variable** *[nombre variable] [valor del fallo]*
3. **Simular Durante** *[duración del fallo]*	3. **Simular Durante** *[tiempo de observación]*
4. **Restaurar Señal** *[nombre señal]*	
5. **Simular Durante** *[tiempo de observación]*	

Tabla 11. Algoritmo para la inyección de fallos mediante el uso de las órdenes del simulador [DGil99a].

Como se puede observar en la tabla anterior, los elementos entre corchetes (a los que denominaremos parámetros) varían para cada fallo inyectado, es decir, son elementos dependientes de un experimento concreto de inyección, mientras que el resto de elementos (lo que hemos llamado órdenes), permanecen fijos. La automatización es muy sencilla utilizando una *macro* con las órdenes fijas a la que se le pasan los parámetros de cada inyección. Como se comentó en el punto 3.2.1.2, no es posible controlar la temporización durante la inyección de fallos en variables.

En la Tabla 11, la columna *Inyección en señales* muestra la inyección de fallos transitorios en señales. Para inyectar fallos permanentes, se deben omiten los pasos 3 y 4, mientras que para inyectar fallos intermitentes, se deberían repetir los pasos del 1 al 4, esperando durante un cierto intervalo de tiempo cada vez que se ejecute el paso 4.

3.3.2 Automatización de los perturbadores

Para realizar esta automatización, en primer lugar se creó una librería de perturbadores. El diseño de estos perturbadores se corresponde con los vistos en la sección 3.2.2.1, concretamente en la Figura 15, y teniendo en cuenta que los perturbadores parametrizables (es decir, aquellos que perturban *buses*), se han adaptado al microprocesador bajo prueba [Gracia00a]. Como se ha

comentado anteriormente, los perturbadores se activan mediante un evento en la señal de control. Una vez que se han añadido los perturbadores al modelo original, la activación de los mismos se realiza mediante ficheros de *macros*, siendo estos fácilmente automatizables. La Tabla 12 muestra el pseudo código de la activación de los perturbadores.

Activación perturbadores
1. **Simular Hasta** *[instante inyección]*
2. **Activar Perturbador** *[control]*
3. **Seleccionar Fallo** *[selección]*
4. **Simular Durante** *[duración del fallo]*
5. **Desactivar Perturbador** *[control]*
6. **Simular Durante** *[tiempo de observación]*

Tabla 12. Algoritmo para la inyección de fallos mediante el uso de perturbadores [Gracia01b].

Como ocurre con la inyección de fallos mediante las órdenes del simulador, los campos entre corchetes dependen de cada inyección, siendo pasados al fichero de *macro* como parámetros. Cabe destacar el uso de la señal de control (pasos 2 y 5) y la señal de selección (paso 3). La señal de control se utiliza para seleccionar uno de los perturbadores presentes en el modelo, siendo esta señal única para cada perturbador. Mediante la señal de selección se elige uno de los posibles fallos a inyectar. Como se ha visto en la sección 3.2.2.1, cada perturbador tiene sus propios modelos de fallos a inyectar, con lo que el tipo de fallo que se inyecta en cada inyección depende del perturbador.

Con respecto a la temporización de los fallos, el algoritmo mostrado en la Tabla 12 se utiliza para inyectar fallos transitorios. Para inyectar fallos permanentes, se deben omitir los pasos 4 y 5, mientras que para inyectar fallos intermitentes, se pueden ejecutar los pasos del 1 al 5 varias veces con diferentes intervalos de tiempo entre las distintas ejecuciones. Más información sobre el proceso de automatización e inserción de perturbadores puede verse en [Baraza03].

3.3.3 Automatización de los mutantes

Como se ha visto en la sección 3.2.2.2, la implementación de mutantes que inyectan fallos permanentes es relativamente fácil. Sin embargo, la implementación de mutantes para la inyección de fallos transitorios se complica más, aunque éste tipo de fallos son más interesantes.

Como se ha explicado anteriormente, la inyección de fallos transitorios se puede automatizar utilizando para ello comandos del simulador. La Tabla 13 muestra el pseudo código incluido en el fichero de *macro* de la automatización de la inyección de fallos transitorios utilizando los mutantes.

El funcionamiento del fichero de *macro* es el siguiente. Una vez que se han creado los mutantes y los ficheros de configuración correspondientes, se selecciona la arquitectura sin fallos y se simula hasta el instante de inyección (pasos 1 y 2). A continuación se almacena el estado de esta simulación sin fallos para poder escribirlo en la arquitectura mutada (pasos 3 y 4). Con esta escritura se consigue que la arquitectura mutada inyecte el fallo a partir de un estado sin fallos (simulado con la arquitectura sin fallos). A continuación, se selecciona la arquitectura mutada (paso 5) y dependiendo de la duración del fallo, el proceso a seguir varía ligeramente.

En el caso de inyección de fallos transitorios, los pasos a seguir serían los siguientes: en primer lugar, se simula durante la duración del fallo (paso 6). A continuación se almacena el estado con el fallo resultante de la mutación para escribirlo en la arquitectura sin fallos (pasos 7 y 8). Para finalizar la inyección, se selecciona la arquitectura sin fallos (no mutada) y se simula durante el tiempo de observación (pasos 9 y 10). Hay que recordar que ésta última selección de la arquitectura sin fallos (paso 9) provoca que la arquitectura simulada sea ésta pero con el estado resultante de la inyección del fallo mediante la mutación.

Activación mutantes
1. **Señal de Guarda <= 1**
2. **Simular Hasta** *[instante inyección]*
3. **Guardar Estado**
4. **Escribir Estado en Arquitectura Mutada**
5. **Señal de Guarda <= 2**
6. **Simular Durante** *[duración del fallo]*
7. **Guardar Estado**
8. **Escribir Estado en Arquitectura sin Fallos**
9. **Señal de Guarda <= 1**
10. **Simular Durante** *[tiempo de observación]*

Tabla 13. Algoritmo para la inyección de fallos mediante el uso de mutantes [Gracia01b].

Si se pretende inyectar un fallo permanente, simplemente hay que prescindir de los pasos 6 al 9. Es decir, una vez que se ha escrito el estado de la arquitectura sin fallos en la arquitectura mutada (paso 4), se selecciona la arquitectura mutada (paso 5) y se simula durante el tiempo de observación (paso 10). En este caso, lo que tenemos es un fallo permanente inyectado en un instante dado de la simulación. Para inyectar fallos intermitentes se deben ejecutar los pasos del 1 al 8 un número aleatorio de veces.

Hay que recordar que todos los pseudo comandos están basados en comandos del simulador. Así pues, los elementos entre corchetes de la Tabla 13 pueden ser leídos por la *macro* de inyección como parámetros, automatizándose así la inyección del fallo.

Respecto a la mejora propuesta en [Baraza03], en este mismo trabajo se comenta cómo sería la mutación, que básicamente consistiría en el algoritmo propuesto para los perturbadores, ya que cada mutación está dirigida por una señal de control, tal y como ocurre con los perturbadores. El pseudo código que se debería incluir en el fichero de *macro* para realizar la automatización de la inyección de fallos transitorios sería el mostrado en la Tabla 13, cambiando la activación de la señal de control de los perturbadores por la activación de la señal de control de los mutantes, tal y como se puede ver en la Tabla 14.

Activación mutantes
1. **Simular Hasta** *[instante inyección]*
2. **Activar Mutante** *[señal de control mutante i]*
3. **Simular Durante** *[duración del fallo]*
4. **Desactivar Mutante** *[señal de control mutante i]*
5. **Simular Durante** *[tiempo de observación]*

Tabla 14. Activación de los mutantes definidos en [Baraza03]

3.3.4 Resumen de los modelos de fallos

En la Tabla 15 se muestra un resumen de los distintos modelos de fallos que se pueden inyectar con cada técnica, en la que hay que destacar la ampliación que se ha hecho respecto de los modelos clásicos (*bit-flip* para fallos transitorios y *stuck-at* para fallos permanentes).

En el punto 4.3, “Modelos de fallos”, se describirán con mayor detalle el significado y los mecanismos físicos relacionados con estos modelos de fallos

Técnica de Inyección	Fallos Transitorios	Fallos Permanentes
Comandos del Simulador	*Pulse**, *Bit-flip***, *Delay* e Indeterminación	*Stuck-at* (‘0’, ‘1’), Indeterminación, *Open-Line* y *Delay*
Perturbadores	*Pulse**, *Bit-flip***, *Delay* e Indeterminación	*Stuck-at* (‘0’, ‘1’), Indeterminación, *Open-Line*, *Delay*, *Short*, *Bridge* y *Stuck-open*
Mutantes	Cambios sintácticos del lenguaje	Cambios sintácticos del lenguaje
* En lógica combinacional ** En elementos de memoria		

Tabla 15. Modelos de fallos aplicables con cada técnica de inyección [Gracia00a].

3.4 Comparación de las técnicas de inyección

La siguiente tabla [Gracia00b] muestra de forma resumida las principales diferencias entre las técnicas implementadas en la presente tesis: órdenes del simulador, perturbadores y mutantes.

	Ventajas	Desventajas
Órdenes del simulador	Facilidad de implementación. No es necesario modificar el código VHDL. Sobrecarga del simulador independiente del sistema.	Dependencia total del simulador. Conjunto de fallos a inyectar depende de las capacidades del simulador.
Perturbadores	Método simple de modificación del modelo VHDL. Son componentes reutilizables. Presenta una mayor capacidad de modelización de fallos que la técnica anterior.	Visión restringida del modelo. Requieren creación, inclusión en el modelo y recompilación del mismo. Hay que añadir un cierto número de señales de control al modelo (activación del perturbador y selección del modelo de fallo a inyectar). La sobrecarga del simulador aumenta con el número de perturbadores.
Mutantes	Se diseñan utilizando toda la potencia del VHDL. Se pueden incluir fácilmente en el modelo. Son componentes reutilizables. Es la técnica que presenta una mayor capacidad de modelización de fallos, siendo estos modelos independientes de la tecnología a utilizar.	Requieren creación, inclusión en el modelo y recompilación del mismo. Cuanto más complejos sean los mutantes, mayor es la sobrecarga que se añade al simulador. Dificultad a la hora de definir modelos de fallos representativos. Los mutantes que inyectan fallos transitorios son difíciles de implementar. Esta técnica presenta la mayor sobrecarga temporal de las tres.

Tabla 16. Principales ventajas y desventajas de la inyección de fallos basada en VHDL [Gracia00b].

Algunas conclusiones interesantes que se pueden extraer de la tabla anterior son:

- Si no se necesitan modelos de fallos muy complejos, la recomendación es utilizar la técnica de las órdenes del simulador. Sin embargo, si se pretende aplicar un conjunto de modelos de fallos más completo, las técnicas de los perturbadores y de los mutantes son las más adecuadas, dependiendo del tipo de modelo a evaluar (perturbadores en modelos estructurales y mutantes en modelos comportamentales).
- Como se verá en el capítulo siguiente, la técnica de los comandos del simulador es la que menor tiempo consume, mientras que la técnica de los mutantes es la que presenta un mayor coste temporal.
- Sin embargo, los perturbadores provocan que la traza de inyección sea mayor que con las otras dos técnicas.
- Las órdenes del simulador es la técnica que menor coste de elaboración presenta, ya que no requiere la modificación del modelo. Por el contrario, los perturbadores y mutantes requieren un coste (tanto espacial como temporal) más elevado para la elaboración del modelo de inyección.

3.5 Resumen. Conclusiones y líneas abiertas de investigación

En este capítulo se han presentado tres técnicas de inyección de fallos en modelos VHDL: órdenes del simulador, perturbadores y mutantes. De las tres técnicas se ha realizado una descripción teórica del funcionamiento de las mismas, explicando cómo funcionan para pasar a continuación a la descripción de la automatización de las mismas. En el siguiente capítulo se describirá el uso de estas técnicas en diferentes experimentos de inyección.

Durante la descripción teórica de las distintas técnicas, diferentes eventos se han puesto de relieve. En la técnica de los comandos del simulador se comenta la posibilidad de inyectar nuevos modelos de fallos, superando los clásicos modelos de fallos utilizados hasta ahora (*bit-flip* para fallos transitorios y *stuck-at* para fallos permanentes).

Sin embargo, hay que remarcar dos hechos del desarrollo de la presente tesis. En primer lugar, se ha implementado un conjunto de perturbadores los cuales amplían los modelos de perturbadores utilizados hasta este momento. Este nuevo conjunto de perturbadores presentan dos características diferenciadoras, como son la extensión de los modelos de fallos inyectados hasta ahora, y la aportación de nuevos diseños de perturbadores, ideados especialmente para trabajar en *buses* y en ambas direcciones (bidireccionales). Y en segundo lugar, se han podido inyectar fallos transitorios utilizando la técnica de los mutantes, solventado una serie de problemas que provocaban que este tipo de fallos no se hubiesen utilizado antes con esta técnica.

En el apartado de la automatización de las diferentes técnicas se describen las *macros* (o ficheros especiales entendibles por el simulador) utilizados para llevar a cabo de forma práctica los pasos descritos de forma teórica anteriormente, terminando este capítulo con una comparativa de las diferentes técnicas.

Un problema abierto en este capítulo es el estudio de la representatividad de los fallos, especialmente en el caso de las técnicas de los perturbadores y de los mutantes. Un primer trabajo realizado sobre representatividad de fallos, aunque utilizando únicamente la técnica de los comandos del simulador se puede ver en [Gracia02a].

En cuanto a los diferentes costes, la técnica basada en las órdenes del simulador es la más sencilla de implementar, aunque depende totalmente del simulador empleado. Además, el conjunto de modelos de fallos es más restringido. Por el contrario, los perturbadores y los mutantes son técnicas más complejas de llevar a la práctica. En el caso de los perturbadores, los ficheros de traza son mayores que los generados por las otras dos técnicas, mientras que en el caso de los mutantes, el coste temporal de la simulación de la inyección de un fallo transitorio es mucho mayor que con cualquiera de las otras dos técnicas. Sin embargo, estas dos técnicas permiten inyectar un conjunto de fallos más amplio.

Otro aspecto que queda abierto es la inclusión de los perturbadores y mutantes en una herramienta de inyección de fallos. Los primeros pasos para esta automatización ya se han realizado (como se puede ver en el capítulo 3.3 “Automatización de las técnicas de inyección en modelos en VHDL” y en [Baraza03]).

4 VFIT: La herramienta de inyección de fallos. Modelos de fallos

4.1 Introducción

Dos aspectos importantes en la realización de los experimentos de inyección son la automatización de los experimentos y la representatividad de los resultados. En este capítulo se explicarán brevemente los métodos utilizados para conseguir ambos aspectos. En primer lugar se hará un resumen de VFIT, la herramienta de inyección de fallos utilizada durante los experimentos de inyección. Con esta herramienta, se consiguió automatizar todo el proceso de inyección, desde la configuración del simulador VHDL hasta el análisis y obtención de los resultados.

En segundo lugar, se comentarán brevemente los modelos de fallos utilizados. Con este estudio se intentó que los modelos de fallos fuesen tan reales como permitiese el simulador VHDL, con el fin de poder obtener unos resultados lo más aproximado que se pudiera con la realidad.

No es el propósito de este capítulo, ni de esta tesis, el explicar el desarrollo de VFIT, o la explicación detallada de la obtención de los distintos modelos de fallos. Información más exhaustiva sobre este tema se puede ver en [Baraza02, DBENCH02, Gracia02a, Gracia02c, Baraza03].

4.2 VFIT: VHDL-Based Fault Injection Tool

En este punto se va a presentar la herramienta utilizada durante los experimentos de inyección de fallos realizados a lo largo de esta tesis. Esta herramienta se denomina VFIT (***V**HDL-Based **F**ault **I**njection **T**ool*) [Baraza00, Baraza02, Gracia02c, VFIT02, DGil03b], y ha sido diseñada e implementada en torno a un simulador comercial de VHDL [Model01a, Model01b]. Con VFIT se pueden inyectar varios modelos de fallos, los cuales se verán en el punto 4.3, con distintas duraciones. Otro aspecto remarcable de esta herramienta es la posibilidad de analizar automáticamente los resultados obtenidos en las distintas campañas de inyección. A partir de estos análisis, se puede estudiar el Síndrome de Error del sistema modelado o validar los Mecanismos de Tolerancia a Fallos del mismo.

4.2.1 Características generales de VFIT

VFIT puede ser ejecutada en una plataforma IBM-PC (o compatible), funcionando bajo el entorno *Windows™*. Como se acaba de comentar, la herramienta ha sido diseñada utilizando como base un simulador comercial de VHDL. Es fácilmente transportable y utilizable, además de permitir campañas de inyección de fallos en sistemas modelados en VHDL de cualquier complejidad. Algunas características significativas son:

- Una única aplicación realiza automáticamente todas las fases de la inyección.
- Universal, fácil de usar.
- VFIT permite inyectar un conjunto muy amplio de modelos de fallos, extendiendo los clásicos modelos de *stuck-at* y *bit-flip*. Con respecto a los modelos de fallos utilizados, éstos dependen de la técnica utilizada y del nivel de abstracción del modelo del sistema. Principalmente, se ha trabajado con circuitos modelados en los niveles de puertas, registros y chip. La Tabla 15 muestra los diferentes modelos de fallos utilizados en cada técnica de inyección. Además, en el punto 4.3 se puede ver un resumen de la obtención de estos modelos de fallos.

- Respecto a la temporización de los fallos, se pueden inyectar fallos transitorios, permanentes e intermitentes, siendo posible elegir entre diferentes distribuciones verificadas en fallos reales (*Uniforme*, *Exponencial*, *Weibull* y *Gausiana*) a la hora de determinar el instante de inyección y la duración del fallo.
- Permite utilizar diferentes técnicas de inyección en VHDL: órdenes del simulador, perturbadores y mutantes.
- VFIT puede realizar dos tipos de análisis:
  - ⇨ Cuando se analiza el *síndrome de error*, se obtiene una clasificación de los fallos y errores, así como su incidencia relativa, calculándose las latencias de propagación para cada tipo de fallo especificado en una clasificación de fallos introducida durante la especificación de los parámetros de análisis. Este tipo de análisis es interesante a la hora de determinar los mecanismos de detección y recuperación más apropiados con el fin de mejorar la Confiabilidad de un sistema.
  - ⇨ Cuando se *valida un Sistema Tolerante a Fallos*, se realiza un estudio detallado del funcionamiento de los mecanismos de detección y recuperación del sistema. A partir de este estudio se calculan diferentes parámetros de la Confiabilidad, como las latencias y coberturas de detección y recuperación. Normalmente, un sistema se valida después de haber realizado un análisis del síndrome de error.

4.2.2 Fases de un experimento de inyección

Un experimento de inyección consiste en inyectar un determinado número de fallos en el modelo, según el valor de los parámetros de inyección. Para cada fallo inyectado se analiza el comportamiento del modelo, y al final del experimento, se obtiene el valor de los parámetros especificados.

Una campaña de inyección es un conjunto de experimentos de inyección diferentes, cambiando uno o varios de los parámetros de inyección (tipo y/o duración del fallo, lugar de inyección, etc.) en cada inyección. Un experimento de inyección consiste en tres fases independientes:

1. ***Configuración de la campaña.*** Con la ayuda de una interfaz gráfica, se especifican los parámetros de inyección y análisis:
   - Los parámetros de inyección más importantes son: técnica de inyección (órdenes del simulador, perturbadores, mutantes), número de inyecciones, objetivos de la inyección (módulos, señales, etc.), modelos de fallos para cada lugar de inyección, distribución de los instantes de la inyección del fallo y de la duración del fallo, ciclo de reloj, carga de trabajo del sistema y duración de la simulación.
   - Por otra parte, las condiciones del análisis dependen del tipo de análisis que se realizará. Si es un estudio del síndrome de error, se deben especificar la clasificación de los fallos y establecer las cláusulas para la clasificación de errores. Si se realiza la validación de un sistema tolerante a fallos, se deben especificar las cláusulas de detección y recuperación de errores.
2. ***Simulación.*** En esta fase se realizan dos operaciones. En primer lugar, se generan automáticamente una serie de *macros*[20] para realizar las inyecciones: una *macro* lleva a cabo una simulación sin fallos, mientras que el resto contienen los comandos necesarios para inyectar el número de fallos especificados. En segundo lugar, se ejecutan estas *macros* en el simulador de VHDL, obteniéndose una serie de ficheros de traza: uno con la simulación del sistema sin fallos y *n* ficheros con un fallo inyectado (siendo *n* el número

[20] Como se ha comentado anteriormente, un fichero de *macro* es un fichero especial que contiene comandos interpretables por el simulador

de fallos inyectados). Este es el caso más común cuando se están inyectando fallos simples. Sin embargo, hay que recordar que VFIT es capaz de inyectar también fallos múltiples en una simulación.

3. ***Análisis de los resultados.*** La traza de la simulación sin fallos se compara con las *n* trazas de las simulaciones con fallos, analizando sus diferencias y extrayendo los parámetros de la Confiabilidad del modelo del sistema. Dependiendo del tipo de análisis, el objetivo de la comparación es diferente, realizando las siguientes acciones el algoritmo de análisis:

   - En un *análisis del síndrome de error*, cuando se encuentra una discrepancia, se clasifica el error y el fallo en función de los tipos de error y fallo, calculándose además la latencia de propagación. Una vez que se ha clasificado el error, se comprueba el resultado de la carga de trabajo. Si el resultado es incorrecto, se actualiza el contador de averías. Los resultados típicos que se obtienen del análisis del síndrome de error son los porcentajes de errores efectivos y su latencia, así como los porcentajes de averías.
   - En una *validación de un Sistema Tolerante a Fallos*, si no se encuentra ninguna discrepancia, se considera que el fallo inyectado no se ha activado. Si se encuentra una discrepancia, se busca el cumplimiento de las cláusulas de detección para determinar si el error activado ha sido detectado o no. Si no ha sido detectado, el error activado puede ser no–efectivo si el resultado de la carga de trabajo es correcto, o producir una avería si este resultado es incorrecto. Si el error ha sido detectado, se comprueba si se cumplen las cláusulas de recuperación para determinar si el error ha sido recuperado o no. Al final del análisis, se puede completar el *grafo de predicados de los mecanismos tolerantes a fallos* [Arlat93] (mostrado en la Figura 28). Este diagrama refleja la patología de los fallos, es decir, la evolución de los fallos desde que estos son inyectados hasta que los errores son detectados y, eventualmente, recuperados. A partir del grafo, se pueden calcular las latencias medias de propagación, detección y recuperación, y las coberturas de detección y recuperación.

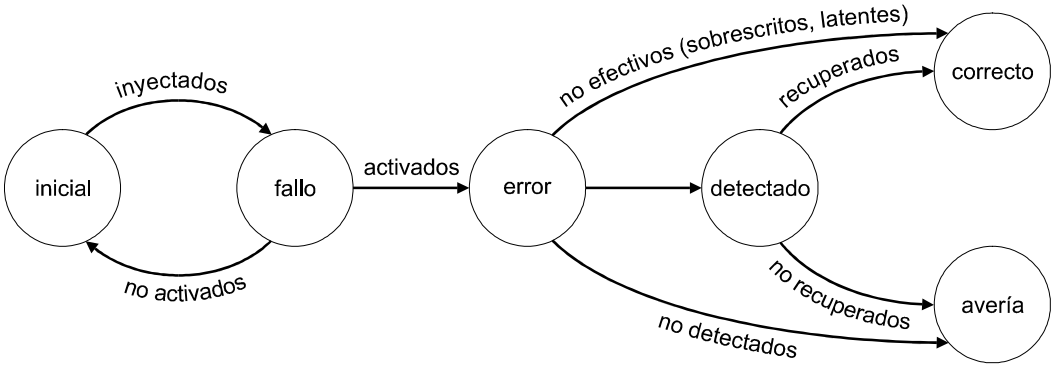


Figura 28. Grafo de predicados de los mecanismos tolerantes a fallos [Arlat93].

4.2.3 Diagrama de bloques de VFIT

La Figura 29 muestra el diagrama de bloques detallado de VFIT. Puede verse como a partir de la interfaz con el usuario, el modelo del sistema bajo estudio y el simulador VHDL, la herramienta es capaz de proporcionar al usuario los resultados de los análisis realizados durante el experimento de inyección.

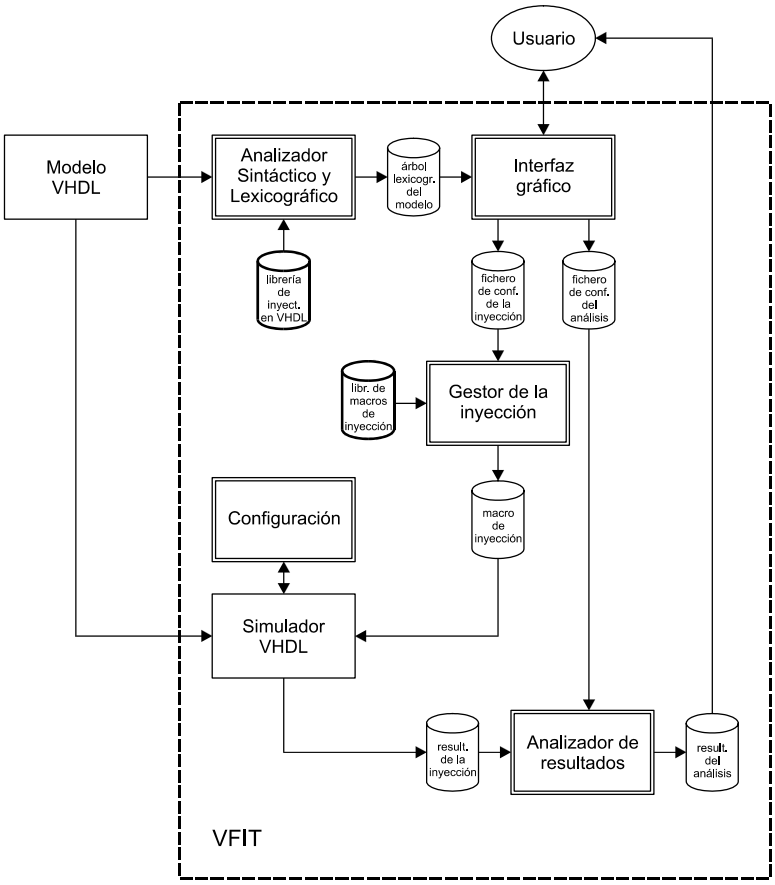


Figura 29. Diagrama de bloques detallado de VFIT [Baraza02].

A continuación se resumen las funciones de los distintos elementos de VFIT.

Configuración

La función de este módulo es configurar a VFIT, estableciendo los parámetros de la herramienta y del simulador, considerando que el simulador es parte de la herramienta. El aspecto más importante es enlazar las librerías usadas.

Analizador Sintáctico y Lexicográfico

La misión de este módulo es el análisis de todos los ficheros del modelo para obtener su *Árbol Sintáctico*. Este árbol incluye, básicamente, todos los posibles puntos del modelo donde es posible realizar una inyección. La estructura en árbol refleja la estructura jerárquica del modelo, incluyendo *componentes*, *bloques* y *procesos*. Dependiendo de la técnica de inyección usada, los posibles puntos de inyección pueden variar:

- **Órdenes del simulador**. Las señales y variables atómicas del modelo del sistema, pertenecientes a las arquitecturas estructurales o comportamentales.
- **Perturbadores**. Las señales atómicas pertenecientes a descripciones estructurales del modelo del sistema.
- **Mutantes**. Los elementos sintácticos de las arquitecturas comportamentales del modelo VHDL.

Librería de inyectores en VHDL

Esta librería mantiene un conjunto de perturbadores VHDL predefinidos que serán incluidos en el modelo cuando esta técnica sea usada. La librería también contendrá los mutantes generados a partir del modelo original.

Interfaz gráfica

Esta utilidad es un amplio conjunto de menús basados en ventanas que permiten al usuario especificar todos los parámetros y condiciones (relacionados con la inyección o el análisis), necesarios para realizar un experimento de inyección. Con estos parámetros y condiciones, se generan los ficheros de *configuración de la inyección* y de *configuración del análisis*.

Librería de *macros* de inyección

Esta librería está compuesta por un conjunto de *macros* predefinidas utilizadas para activar los mecanismos de inyección según la técnica de inyección utilizada. Las *macros* se diseñan a partir de un subconjunto de órdenes del simulador.

Gestor de la inyección

Este módulo controla el proceso de inyección. Utilizando el fichero de *configuración de la inyección* generado por la *interfaz gráfica*, este módulo:

1. Crea una serie de *macros* de inyección, con el fin de realizar una simulación sin fallos y el número de simulaciones con fallos inyectados especificados en los parámetros, e
2. Invoca al simulador para ejecutar estas *macros*, obteniendo las *trazas de simulación*.

Simulador VHDL

Como se ha indicado anteriormente, se ha utilizado el simulador comercial de VHDL *ModelSim*, el cual proporciona un entorno para IBM-PC (o compatibles) bajo *Microsoft Windows*™. Este simulador está dirigido por eventos y es muy simple y fácil de usar. Cuando se activa, el simulador ejecuta el fichero con las *macros* y genera las trazas de salida de la simulación.

Analizador de resultados

Este módulo toma como entradas el fichero de *configuración del análisis* generado por la *interfaz gráfica*. Según estos parámetros, se compara la traza sin fallos con la traza con fallos inyectados, buscando discordancias entre ellas, y extrayendo los parámetros especificados del análisis.

4.3 Modelos de fallos

4.3.1 Introducción

El primer paso a la hora de definir unos modelos de fallos adecuados, es la definición de los niveles de abstracción en los que se puede dividir un sistema informático. Los modelos de fallos que se van a utilizar en la presente tesis están basados en la división de niveles definida en [DBENCH02]. Estos niveles se pueden ver en la Figura 30. Existen otras clasificaciones, como las presentadas en [Walker85, Siewiorek92, Jenn94a, Pradhan96]. En estos casos, todos los niveles están relacionados e incluso se solapan.

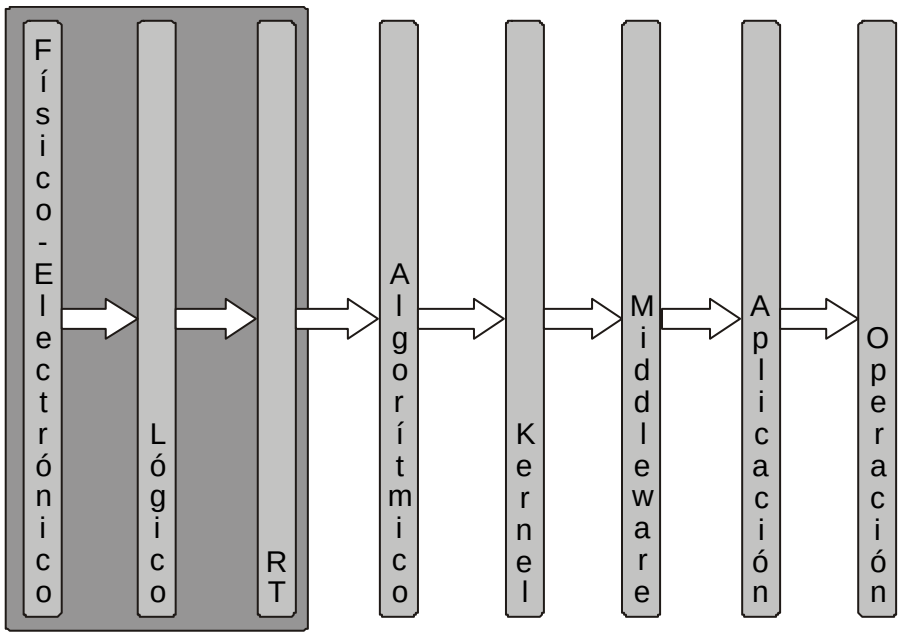


Figura 30. Niveles utilizados en la modelización de los fallos hardware [DBENCH02].

Como es sabido, los fallos que se producen en un nivel pueden propagarse como errores o averías en niveles superiores. No es el propósito de este trabajo la definición de los distintos niveles ni su justificación. En este capítulo simplemente se hará un resumen de los modelos de fallos utilizados durante los diferentes experimentos de inyección presentados a lo largo de esta tesis. Más información sobre este tema puede encontrarse en [DBENCH02, Baraza03].

En función de la Figura 30 podemos decir que los fallos *hardware* abarcan los tres primeros niveles (físico–electrónico, lógico y RT). Las diferentes técnicas de inyección de fallos en VHDL empleadas a lo largo de esta tesis utilizan modelos de fallos centrados en los niveles lógico y RT. Así pues, el resto de este capítulo presentará un resumen de los distintos modelos utilizados en estos dos niveles.

Antes de comenzar, haremos una primera clasificación de los fallos en función de su duración:

- Fallos transitorios, con una duración corta en el tiempo.
- Fallos permanentes, que perduran de manera indefinida.
- Fallos intermitentes, similares a los transitorios en cuanto a que tienen una duración finita, pero que se repiten en el tiempo sin un comportamiento periódico.

4.3.2 Mecanismos de fallos

Tradicionalmente, los modelos de fallos predominantes han sido *stuck-at* (‘0’ y ‘1’) para los fallos permanentes, y *bit-flip* para los fallos transitorios. Sin embargo, en los últimos años, el desarrollo tecnológico submicrónico ha propiciado que el uso exclusivo de dichos modelos en la inyección de fallos no sea representativo de la casuística real. Con el fin de subsanar esta deficiencia, es necesario introducir nuevos modelos que sirvan tanto para los circuitos integrados actuales así como para las futuras tecnologías.

El resto de este capítulo resume los mecanismos físicos implicados en la ocurrencia de cada tipo de fallo, asociándoles una serie de modelos de fallo en los niveles de abstracción lógico y RT [Amerasekera97, DGil99a, DBENCH02, Baraza03].

4.3.2.1 Fallos permanentes

La Figura 31 muestra la casuística de los fallos permanentes más significativos. Estos fallos se deben a defectos irreversibles en los circuitos, producidos durante el proceso de fabricación o por desgaste. Los óvalos muestran los fallos deducidos, mientras que los rectángulos muestran las causas y los mecanismos físicos que los producen [Siewiorek92, Amerasekera97, Pradhan96, DBENCH02, Baraza03].

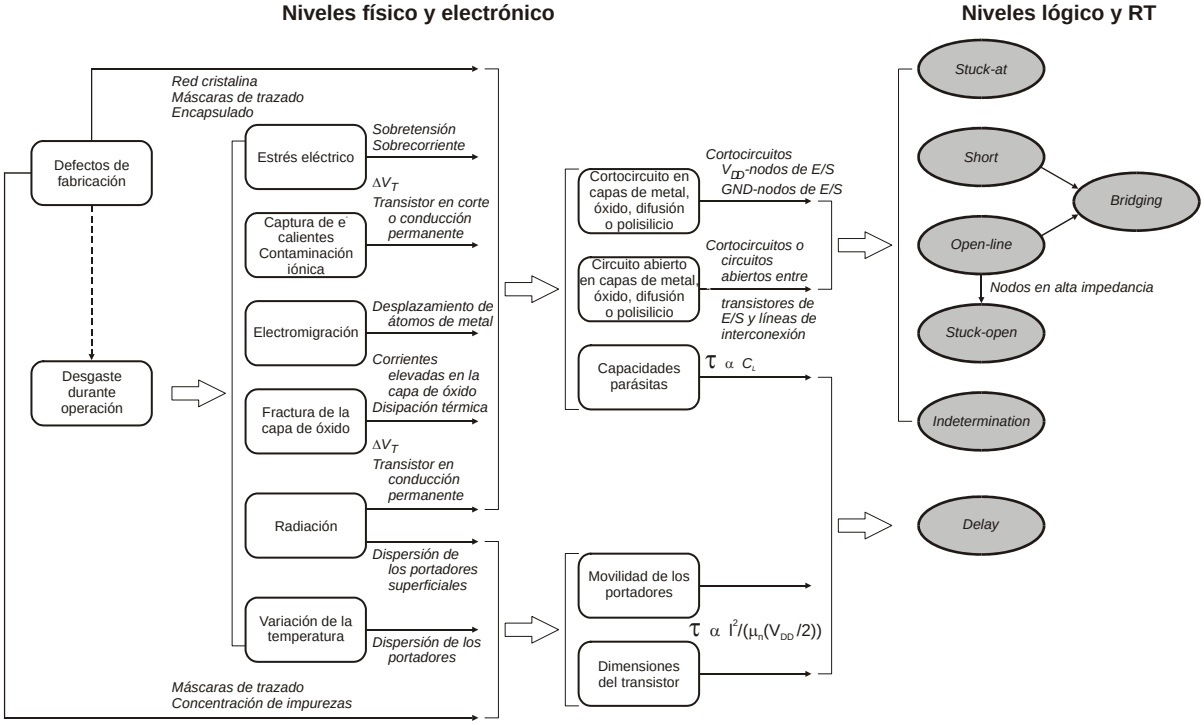


Figura 31. Mecanismos de fallos permanentes y modelos de fallo equivalentes [DBENCH02].

Para deducir los modelos de fallos a nivel físico–electrónico, los mecanismos han sido clasificados en dos grupos. En el primer grupo se consideran los mecanismos que provocan cortocircuitos y circuitos abiertos en las diferentes capas del transistor (metal, óxido, etc.). El efecto de estos fallos en los niveles lógico y RT es un conjunto de modelos de fallo que a veces están relacionados entre sí:

- ***Stuck-at* (‘0’, ‘1’)**. Este es el modelo más comúnmente utilizado.
- ***Open-line*** (circuito abierto, o alta impedancia) en las líneas de conexión de circuitos lógicos.
- ***Short*** (cortocircuito) entre las líneas de conexión de circuitos lógicos.

- ***Bridging***[21] (puente), causado por combinaciones de cortocircuitos y circuitos abiertos.
- ***Stuck-open***. Este fallo se debe a nodos flotantes en alta impedancia, que mantienen el valor lógico del transistor durante un tiempo denominado **tiempo de retención**, que es el tiempo que tardan en descargarse las capacidades parásitas de salida a causa de las corrientes de fuga. Durante el tiempo de retención (del orden de milisegundos [Pradhan86]), el circuito presenta un comportamiento secuencial, hasta que se descarga (quedando entonces un valor lógico '0'). Este tipo de fallo es característico de los circuitos integrados MOS.
- ***Indetermination*** (indeterminación). En este caso, el fallo puede deberse tanto a cortocircuitos en las salidas del circuito lógico como a circuitos abiertos en las entradas.

En el segundo grupo se incluyen algunos mecanismos de fallo que afectan al retardo de conmutación de los transistores MOS y a los tiempos de carga/descarga de las capacidades parásitas en las conexiones de entrada/salida: movilidad de portadores, modificación de las capacidades parásitas y variación de las dimensiones del transistor [DGil99a]. Su efecto en los niveles lógico y RT es una modificación permanente de los retardos de los circuitos lógicos, por lo que se modelan con el fallo denominado ***delay*** (o alteración de los retardos).

4.3.2.2 Fallos intermitentes

Algunos de los mecanismos de fallo por desgaste (que provocan fallos permanentes) pueden manifestarse inicialmente de forma esporádica e intermitente, sin presentar ninguna cadencia concreta de aparición, hasta que el daño se vuelve irreversible. Por este motivo, los modelos de fallos que se pueden aplicar son los mismos que para los fallos permanentes.

4.3.2.3 Fallos transitorios

Estos fallos no introducen ningún defecto físico en el circuito. También se denominan *soft errors* y *single event upsets* (SEU), y pueden aparecer durante el funcionamiento de un circuito por diferentes causas, tanto internas como externas. La Figura 32 muestra los modelos de fallos deducidos para los fallos transitorios. En este caso, los modelos ***bit-flip*** (aplicado a elementos de almacenamiento: memorias y registros), ***pulse***[22] (aplicado a circuitos combinacionales) e ***indetermination*** permiten representar fallos físicos de diferente naturaleza: transitorios en la fuente de alimentación, diafonía (en inglés *crosstalk*), interferencias electromagnéticas (luz, radio, etc.), variaciones de temperatura, radiación (de partículas α y cósmica[23]), etc.

Estos fallos físicos pueden variar los valores de tensión y corriente de los niveles lógicos en los puntos de un circuito, como ocurre por ejemplo a causa de los transitorios en la tensión de alimentación. También pueden provocar la generación de pares electrón–hueco, que son barridos por el campo eléctrico de las zonas de deplexión de las uniones p–n de los transistores. La corriente de pares electrón–hueco generada puede modificar el valor lógico de un nodo del circuito, conmutando su valor. Si el nodo pertenece a una celda de almacenamiento (registros o memorias SRAM o DRAM) [Amerasekera97], el fallo se denomina *bit-flip*. Si pertenece a un circuito combinacional, se puede alterar el nivel lógico, bien para conmutarlo (modelo *pulse*) o para dejarlo indeterminado (modelo *indetermination*).

---

[21] Aunque tradicionalmente el nombre de este modelo hace referencia a un cortocircuito [Pradhan86, Shaw01], algunos autores consideran como puente ciertas combinaciones entre cortocircuitos y circuitos abiertos que dan lugar a fallos más complejos [Jenn94a, DGil99a], y al modelo considerado tradicionalmente como puente se le denomina cortocircuito (*short*).

[22] Pulso.

[23] La radiación cósmica es una fuente muy importante de fallos transitorios en aplicaciones aeronáuticas y espaciales.

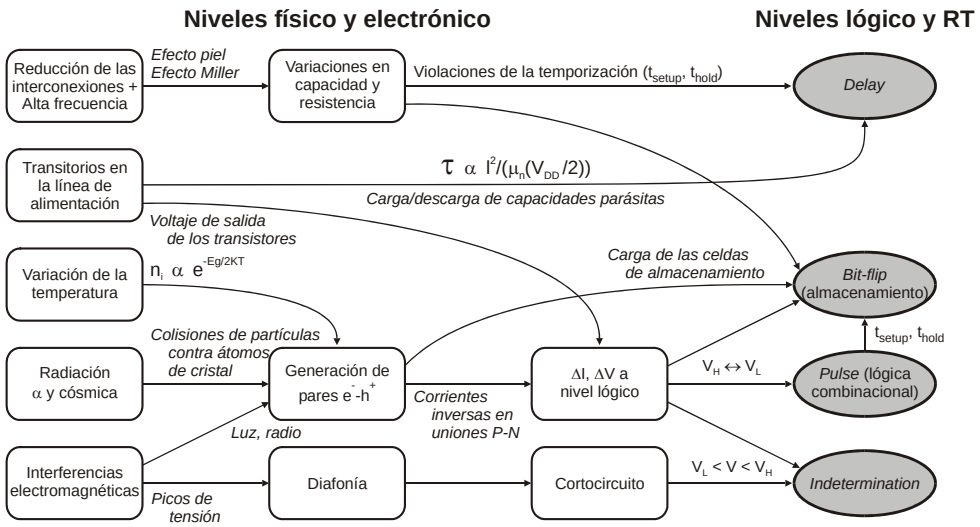


Figura 32. Mecanismos de fallos transitorios y modelos de fallo equivalentes [DBENCH02].

El modelo ***delay*** permite representar fallos físicos debidos a transitorios en la alimentación que pueden alterar el retardo de conmutación de los transistores MOS o los tiempos de carga/descarga de las capacidades parásitas de las conexiones de entrada/salida.

Uno de los problemas principales de los fallos transitorios y, por consiguiente, de sus modelos asociados, es determinar su duración. Sin embargo, muy poco se ha escrito acerca de este asunto [DGil99a].

4.3.3 Influencia de las nuevas tecnologías submicrónicas

A medida que aumenta el uso de los circuitos VLSI en sistemas fiables o de seguridad crítica (del inglés *critical–safety systems*), aumenta la importancia de la Confiabilidad de dichos circuitos. Al reducir las geometrías y las tensiones de alimentación y aumentar las frecuencias de funcionamiento, la Confiabilidad se resiente, ya que se incrementan las tasas de los fallos. A continuación se muestra un resumen del impacto de las nuevas tecnologías submicrónicas en los mecanismos físicos del fallo. De esta manera, se relacionan dichos mecanismos con modelos de fallo en los niveles lógico y RT.

4.3.3.1 Fallos permanentes

Los grupos de mecanismos de fallo más significativos en las nuevas tecnologías son [Baraza03]:

- **Daños en la capa de óxido**.
  - ⇨ Rotura de la capa de óxido (*thin oxide breakdown*). Depende del espesor de la capa de óxido de la puerta. Básicamente, la rotura es un proceso que se produce en dos fases, conocidas como desgaste y rotura [Hu99, Stathis01]. Los efectos pueden ser una excesiva corriente de fuga en los *pines* de entrada y salida, un incremento de la disipación de potencia en el circuito integrado y una disminución de la velocidad del circuito. Los modelos de fallos deducidos son ***short***, ***open-line***, ***indetermination***, ***delay*** y ***bit-flip*** (véase la Figura 31).
  - ⇨ Inyección de portadores calientes (*hot carrier injection*). Habitualmente, la degradación de las prestaciones del transistor se atribuye a la presencia de cargas fijas en la capa de óxido a causa de la captura de electrones y huecos en dicha capa [Rodder95]. El efecto de este mecanismo de fallo es una degradación gradual de algunos parámetros del transistor, como la tensión umbral, la transconductancia y la

corriente del drenador [Hawkins00]. Los modelos de fallos deducidos son ***indetermination*** y ***delay***.

⇨ Daños por plasma. El empleo de plasma[24] puede dañar las capas de óxido de los dispositivos MOS. El plasma incidente acumula carga en los electrodos de metal, habitualmente en las regiones donde el área o la periferia son grandes [Fang93]. Dicha carga se transfiere a la región del correspondiente electrodo, permitiendo un cierto flujo de corriente a través de la fina capa de óxido de la puerta [McVittie96]. El efecto es un incremento en la corriente de fuga de la capa de óxido. Los modelos de fallos deducidos son ***indetermination***, ***delay*** y ***bit-flip***.

- **Metalizaciones**.

⇨ Electromigración. A medida que las conexiones metálicas se hacen más delgadas y aumenta la densidad de corriente, se incrementa el problema de la electromigración [Hawkins00]. El efecto es un incremento de la resistencia de interconexión, e incluso un circuito abierto. La electromigración también puede producir cortocircuitos entre conexiones contiguas (en la misma capa) o situadas en capas adyacentes. A nivel lógico, los modelos que se pueden deducir son [DGil99a]: ***stuck-at***, ***short***, ***open-line*** (en ocasiones ***stuck-open***), ***bridging***, ***indetermination*** y ***delay*** (véase la Figura 31).

⇨ Migración inducida por estrés (*stress voiding*). Durante el proceso de fabricación de la oblea, el encogimiento posterior a la deposición de las diferentes capas puede producir una transferencia de átomos [Jones87, Tezaki90]. Este efecto se manifiesta mediante la formación de huecos (en inglés *voiding*) y de muescas (en inglés *notching*) en las líneas metálicas [Oates93], así como la formación de "rebarbas" metálicas (en inglés *whiskers*) [Turner82], que pueden producir cortocircuitos. Los modelos de fallos a nivel lógico son básicamente los mismos que en el caso anterior: ***stuck-at***, ***short***, ***open-line***, ***stuck-open***, ***bridging***, ***indetermination*** y ***delay***.

⇨ Otros.

- Migración en los contactos (*contact migration*). Este mecanismo está basado en la migración de los átomos de metal y silicio en los contactos metálicos.
- Migración en las vías. Este fallo se da en los circuitos multicapa, en los contactos metal–metal (vías). El mecanismo es similar a la migración en los contactos, pero en este caso sólo se desplazan átomos de metal, desde o hacia la vía.
- Microfracturas y defectos de cobertura de escalones. Las regiones de una oblea con escalones pronunciados son susceptibles a la electromigración por el paso de elevadas densidades de corriente [Pol96], o a circuitos abiertos por la fractura de la metalización. La delgadez de las capas también es un problema en las zonas que rodean los contactos metálicos [Saito93], pues la migración en los contactos se ve favorecida.

Puesto que los efectos de todos estos mecanismos se pueden resumir como circuitos abiertos y cortocircuitos en las interconexiones metálicas [Amerasekera97], los modelos a nivel lógico son los mismos que para la electromigración y la migración por estrés: ***stuck-at***, ***short***, ***open-line***, ***stuck-open***, ***bridging***, ***indetermination*** y ***delay***.

- **Encapsulado y ensamblado**.

⇨ Defectos de fijación del chip (*die attach*). Pueden ocasionar dos tipos de mecanismos de fallo: la aparición de huecos (*voids*) en la capa de adhesivo o la introducción de impurezas o de humedad por parte del adhesivo o la base. En ambos casos se produce la corrosión del chip, que origina otros fallos por desgaste. Los efectos de los mecanismos están relacionados con la fusión (en inglés *burnout*) del chip, variaciones

[24] Gas ionizado.

paramétricas o la corrosión. Además, se puede producir la rotura del chip durante procesos con excesivo estrés mecánico o térmico.

- ⇨ Defectos de conexión. La conexión del chip con los *pines* se realiza mediante un cable y dos conexiones: una con el chip y otra con el soporte de los *pines*. Si el cable sufre tensiones (mecánicas) fuertes, la unión con el anclaje puede romperse, provocando circuitos abiertos. Existen varios puntos débiles: i) Si el cable es demasiado estrecho, tensiones fuertes pueden provocar la fractura del cable. ii) Si, por el contrario, la tensión es demasiado débil, el cable puede moverse, provocando cortocircuitos con cables adyacentes. iii) Dependiendo de las impurezas y la humedad existentes, se pueden generar aleaciones indeseables que pueden debilitar las uniones, así como poros. Todos estos problemas se traducen en la ruptura de las uniones, provocando circuitos abiertos. El efecto más común son los circuitos abiertos debidos a desplazamientos en las conexiones con el chip.
- ⇨ Deslaminación y efecto "palomita de maíz" (*popcorn effect*). La humedad absorbida por un circuito integrado puede expandirse en posteriores procesos a alta temperatura y ocasionar varios fallos. Entre ellos, son característicos la deslaminación entre el chip y el soporte y entre el soporte y el encapsulado (rompiendo el chip). La degradación de los cables de conexión, la rotura del encapsulado, el desplazamiento de la metalización y la corrosión pueden ocasionar fallos eléctricos debidos al incremento de las corrientes de fuga, fallos intermitentes o circuitos abiertos.
- ⇨ Corrosión. Cuando llega humedad al chip, ésta actúa como catalizador del mecanismo de corrosión en las metalizaciones. Los principales mecanismos de fallo consisten en un incremento en la resistencia de las metalizaciones, que en ocasiones conducen a circuitos abiertos. A causa de la migración, o por la aparición de dendritas, también se puede producir un aumento en las corrientes de fuga entre líneas de metalización adyacentes, dando lugar a cortocircuitos.

En todos los casos, los modelos asociados son circuitos abiertos o cortocircuitos en las líneas, siendo los modelos propuestos básicamente los mismos que los presentados en la electromigración.

4.3.3.2 Fallos intermitentes

Sobre este tipo de fallos hay que tener en cuenta que son fallos que ocurren repetidamente en el mismo circuito, eliminándose el fallo cuando se repara el circuito afectado. Estos fallos tienen mayores tasas de fallo[25] que los fallos transitorios, y son la manifestación inicial de los fallos permanentes. Es decir, tienen su origen en defectos físicos y procesos de desgaste de los circuitos integrados. Por último, las elevadas frecuencias de funcionamiento aumentan el impacto de las indeterminaciones temporales, que provocan violaciones de los márgenes temporales de seguridad y como consecuencia, fallos intermitentes [Constantinescu01, Constantinescu02, Shivakumar02].

Los modelos de fallos son similares a los de los fallos permanentes puesto que la mayoría de los mecanismos involucrados coinciden. Se pueden utilizar los mismos modelos (***stuck-at***, ***short***, ***open-line***, ***stuck-open***, ***bridging***, ***indetermination*** y ***delay***), pero considerando que su aparición y duración no tienen una cadencia determinada (es decir, son aleatorios).

4.3.3.3 Fallos transitorios

Como ya se ha comentado, este tipo de fallos afecta al sistema durante un intervalo de tiempo relativamente corto, no introduciendo ningún defecto físico en el circuito. Estos fallos son muy problemáticos de tratar debido a su corta duración y a la imposibilidad de conocer la

[25] Número de fallos por unidad de tiempo.

situación exacta del mismo antes de la ocurrencia del fallo. Tradicionalmente, el modelo básico utilizado ha sido el ***bit-flip***.

Algunos aspectos que más han influido en la reducción de las geometrías de los nuevos transistores submicrónicos son también la causa (directa o indirecta) del aumento de la tasa de fallos transitorios. Por ejemplo en [Shivakumar02] se mencionan: i) la disminución de la tensión de alimentación, ii) el aumento de la velocidad de conmutación, iii) el aumento de las etapas de segmentación (en inglés *pipelining*), iv) la disminución de la carga crítica de las celdas, etc. Como consecuencia, actualmente se producen más fallos transitorios, y además, se producen en lugares en los que no se consideraban los modelos de fallos, como pueden ser los circuitos combinacionales.

Además, y debido a la reducción de las geometrías en los circuitos integrados actuales, aumenta la probabilidad de ocurrencia de fallos múltiples, ya que la energía necesaria para provocar un cambio de valor disminuye.

4.3.3.3.1 Radiación de partículas α

Normalmente, este mecanismo de fallo se ha asociado a las celdas DRAM y a los registros dinámicos, modelándose como un fallo de tipo ***bit-flip***. Sin embargo, tal como se ha comentado, los fallos ahora pueden aparecer en circuitos combinacionales, no siendo válido en este caso el modelo del *bit-flip*, ya que el efecto del fallo es distinto: cuando el efecto del impacto desaparece, el transistor recupera el estado correcto, ya que los circuitos combinacionales carecen de realimentación, por lo que no se "memorizará" el valor erróneo de tensión. Por eso, se ha definido un nuevo modelo de fallo denominado ***pulse***, que asume esta diferencia con el *bit-flip*, y que sólo es aplicable a elementos combinacionales. Esta característica es extensible a todos los fallos transitorios debidos a radiación, independientemente de su tipo.

Además, también es posible considerar otros modelos. Por ejemplo, si la carga generada es próxima a la carga crítica, el cambio en la tensión puede producir una salida indeterminada (modelo ***indetermination***) [DGil99a].

4.3.3.3.2 Radiación de rayos cósmicos

Cuando los rayos cósmicos entran en la atmósfera, colisionan con átomos atmosféricos, produciendo partículas cósmicas: fotones, electrones, protones, neutrones, piones, etc. De entre estas partículas cósmicas, tradicionalmente se ha considerado a los neutrones de alta energía (superior a 1 MeV) como la principal fuente de fallos transitorios en dispositivos CMOS. Como en el caso anterior, el modelo más utilizado para representar los efectos de este mecanismo de fallo es el ***bit-flip***. Como ya se comentó en el apartado anterior, también hay que considerar el modelo ***pulse*** para elementos combinacionales, así como el modelo ***indetermination***.

4.3.3.3.3 Otros mecanismos

Además de la radiación, hay que tener en cuenta otras causas de fallos debidas al funcionamiento con frecuencias elevadas, ligadas sobretodo al incremento de la resistencia de las conexiones metálicas y de la capacidad parásita entre las líneas de conexión, afectando al retardo de propagación de las señales, como por ejemplo, el efecto piel (del inglés *skin effect*) [Walker00] y el efecto *Miller* [Sylvester99]. Dichas violaciones de los márgenes temporales de seguridad (que se modelan con el fallo ***delay***) pueden corromper los datos transferidos, provocando fallos de tipo ***bit-flip*** o ***indetermination*** en los elementos de almacenamiento.

Otros mecanismos que pueden provocar fallos transitorios, además de la radiación o los retardos en las interconexiones metálicas, son las variaciones transitorias de la tensión de alimentación y las interferencias electromagnéticas internas (diafonía, o *crosstalk*) o externas [DGil99a]. Estos mecanismos también pueden originar fallos de tipo ***bit-flip***, ***pulse***, ***delay*** e ***indetermination***, como se puede ver en la Figura 32.

4.3.4 Resumen y conclusiones de los modelos de fallos

En este apartado se ha presentado un resumen de los principales mecanismos de fallo en las tecnologías submicrónicas actuales, ampliándose los modelos de fallos utilizados tradicionalmente en los experimentos de inyección de fallos (*stuck-at* para los fallos permanentes y *bit-flip* para los transitorios). Estos modelos se han derivado para los niveles lógico y de transferencia de registro (RT) y para fallos permanentes, intermitentes y transitorios.

Nuevos mecanismos de fallos permanentes e intermitentes se ven potenciados por la reducción de las geometrías. Estos mecanismos afectan principalmente a la capa de óxido de los transistores, las conexiones y el encapsulado de los circuitos integrados. Los fallos se manifiestan básicamente, a nivel electrónico, como cortocircuitos y circuitos abiertos, y que a nivel lógico se pueden modelar como fallos de tipo *stuck-at*, *open-line*, *stuck-open*, *indetermination*, *delay*, *short* y *bridging*, esperándose además un gran aumento de la tasa de fallos intermitentes.

En cuanto a los fallos transitorios, dos hechos han provocado un aumento de la tasa de fallos: la reducción de las geometrías y de las tensiones de alimentación, que junto con el aumento de las frecuencias de funcionamiento, han causado una notable reducción del efecto de los mecanismos naturales de enmascaramiento de los circuitos digitales. Por otra parte, la reducción de las distancias entre las conexiones de los chips provocan la aparición de efectos capacitivos que afectan a la respuesta temporal de los circuitos.

Otra fuente de errores son las radiaciones, tanto internas como externas, afectando también a puntos que tradicionalmente no se tenían en cuenta, como la lógica combinacional. Otro efecto muy importante debido a la reducción de la carga crítica es el aumento de la probabilidad de aparición de fallos múltiples por radiación (sobre todo por rayos cósmicos de alta energía).

Además del conocido *bit-flip*, se introducen otros modelos de fallos que pueden representar los fallos causados por radiación: *pulse* e *indetermination*. Aunque los modelos *bit-flip* y *pulse* son muy parecidos, hay que diferenciar la respuesta del circuito. El modelo *bit-flip* se aplica a elementos de almacenamiento, e implica que el efecto del fallo permanece hasta que se almacena un nuevo valor. Por el contrario, el modelo *pulse* se aplica a lógica combinacional, e implica que el efecto del fallo desaparece con el propio fallo.

El modelo *indetermination* está relacionado con violaciones de los márgenes temporales de los elementos de almacenamiento, y con valores intermedios de tensión en los nodos combinacionales y secuenciales.

Otro modelo propuesto es el *delay*, que representa la alteración de los retardos de propagación de los circuitos debidos a los diferentes efectos capacitivos y resistivos que se producen en las metalizaciones de las tecnologías submicrónicas.

Para finalizar, hay que especificar que en este apartado no se ha intentado hacer un estudio exhaustivo sobre los distintos mecanismos de fallos, nuevas tecnologías, etc. Simplemente, se ha querido hacer un resumen de los modelos de fallos que implementa VFIT, la herramienta de inyección de fallos utilizada en los distintos experimentos de inyección llevados a cabo durante esta tesis. Más información sobre modelos de fallos puede encontrarse en [DBENCH02, Baraza03].

4.4 Resumen. Conclusiones y líneas abiertas de investigación

En este capítulo se ha presentado VFIT, la herramienta de inyección de fallos utilizada durante los experimentos de inyección realizados a lo largo de esta tesis, así como los modelos de fallos que puede inyectar esta herramienta.

Respecto a VFIT, las características más significativas son:

- Todas las fases de la inyección son ejecutadas automáticamente por una única aplicación.
- Se trata de una herramienta completa, de manera que puede:
  - ⇨ Implementar diferentes técnicas de inyección: órdenes del simulador, perturbadores y mutantes.
  - ⇨ Inyectar un amplio conjunto de modelos de fallos.
  - ⇨ Inyectar fallos con distintas temporizaciones: transitorios, permanentes e intermitentes.
- Presenta una interfaz gráfica simple y fácil de usar.
- VFIT puede realizar dos tipos de análisis:
  - ⇨ Síndrome de error.
  - ⇨ Validación de un Sistema Tolerante a Fallos.

Cabe destacar que durante el desarrollo de la presente tesis, se implementó el módulo analizador de VFIT. Este módulo fue mejorado y se le añadieron nuevas prestaciones a medida que los modelos utilizados durante los diferentes experimentos de inyección de fallos se volvían más complejos.

Además, se han propuesto varias líneas para mejorar el funcionamiento y las prestaciones de VFIT. En primer lugar, actualmente se está implementando la inclusión de las técnicas de los perturbadores y mutantes, cuyos estudios previos se han presentado en capítulos anteriores.

En segundo lugar, se está diseñando una versión distribuida de la herramienta. Esta nueva versión permitirá utilizar una red de ordenadores para realizar las distintas simulaciones así como efectuar un análisis también distribuido. Una vez que todos los ficheros estén analizados, un *analizador central* fusionará los resultados y se los proporcionará al usuario. Obviamente, todo el proceso de distribución será transparente al usuario.

En cuanto a los modelos de fallos, se pretende ampliar algunos de los aspectos presentados en este capítulo. Por un lado, la utilización de herramientas de simulación electrónica para estudiar el efecto de los fallos transitorios en nuevas tecnologías, y por otra parte el análisis de las características temporales de los fallos transitorios, en particular su duración.

Otro tema a desarrollar es el estudio de la representatividad de los modelos de fallos propuestos en el nivel RT y superiores (algorítmico, etc.). Por ejemplo, utilizando inyección de fallos sobre modelos en VHDL se pueden inyectar fallos en modelos estructurales de sistemas a nivel lógico (o de puerta) para ver cómo se manifiestan los errores propagados en el nivel RT. Un primer trabajo se presentó en [Gracia02a].

5 Aplicación de nuevas técnicas de inyección de fallos en modelos VHDL

5.1 Introducción

En el capítulo 3, "Técnicas de inyección de fallos basadas en VHDL", se ha profundizado en el estudio de tres técnicas de inyección de fallos en modelos VHDL: órdenes del simulador, perturbadores y mutantes. Para probarlas, estas técnicas se han aplicado sobre un modelo de un microprocesador tolerante a fallos. A continuación se describen estos experimentos de inyección de fallos.

5.2 Experimentos de inyección de fallos

En experimentos iniciales de inyección de fallos se utilizó la técnica de las órdenes del simulador para realizar el análisis del síndrome de error de un microprocesador y la validación de la Confiabilidad de un sistema microprocesador tolerante a fallos. Se comenzó con el análisis del síndrome de error sobre un sistema no tolerante a fallos. Este sistema (denominado MARK2 [Armstrong89]) estaba compuesto por un microprocesador de 8 bits, memoria RAM, un puerto paralelo de entrada y otro de salida, una UART (puerto paralelo–serie/serie–paralelo) y un controlador de interrupciones. Durante estos experimentos, se calcularon los siguientes datos [DGil98a]:

- Clasificación de los fallos.
- Clasificación de los errores efectivos.
- Latencias de propagación,
- Análisis de la influencia de la duración del fallo, el lugar de la inyección y la distribución de los fallos.

Los parámetros de esta campaña de inyección fueron:

1. **Técnica de inyección**: Órdenes del simulador.
2. **Número de fallos**: n = 3000 fallos simples por campaña.
3. **Carga de trabajo**: Serie aritmética de j números enteros:

$$serie = \sum_{k=1}^{j} k \text{ , } j=6 \qquad (1)$$

4. **Tipos de fallos**: Transitorios de tipo *Stuck-at* ('0', '1') y *Open-line* (alta impedancia).
5. **Lugar de la inyección**: Los fallos han sido sistemáticamente inyectados en todas las señales y variables del modelo.
6. **Instante de inyección**: Seleccionado según las distribuciones *uniforme*, *exponencial* y *Weibull* en el rango [0, $t_{Workload}$], donde $t_{Workload}$ es la duración de la carga de trabajo sin fallos.
7. **Duración de la simulación**: El tiempo total de simulación incluye el tiempo de ejecución de la carga de trabajo, más el tiempo de recuperación de la CPU de repuesto ($t_{Simulación}$ = $t_{Workload}$ + $t_{CPU\text{-}Repuesto}$).
8. **Duración de los fallos**: Se han generado con una duración de 0.1T, 0.2T, 0.3T, 0.4T, 0.5T, 1.0T, 1.5T, 2.0T, donde T es el ciclo de reloj de la CPU. Se han inyectado fallos cortos, con una duración igual a una fracción del período del reloj (los más comunes según [Cha93]).

9. **Resultados de los análisis**: Para cada experimento se han recogido los siguientes datos:
    1. Latencia de propagación (tiempo transcurrido desde la inyección del fallo hasta que el error se manifiesta en las señales externas).
    2. Fallo(s) en señal(es) o variabl(es).
    3. Clasificación del tipo de fallo.
    4. Tipo de error.

Estos experimentos están profusamente descritos en [DGil98a, DGil99a], por lo que no se incidirá en los mismos. Sin embargo, a partir de los resultados obtenidos, se introdujeron diferentes mecanismos de detección y corrección de errores para configurar un sistema tolerante a fallos.

Así pues, el siguiente paso fue la validación del sistema anterior. El nuevo sistema era una versión mejorada del MARK2, al que se le añadieron diferentes mecanismos para aumentar la Confiabilidad del mismo. Los mecanismos de detección de errores incluían bit de paridad y un control del flujo del programa mediante un temporizador de guardia. Los mecanismos de recuperación de errores comprenden la introducción de la re–ejecución de la instrucción anterior (en inglés *Back–off cycle*) cuando el error es detectado por el mecanismo de paridad, puntos de recuperación (en inglés *Checkpointing*) cuando los errores son detectados por el temporizador de guardia y un procesador de repuesto (en inglés *Stand–by Sparing*) cuando los errores son permanentes [DGil99b].

El propósito de los diferentes experimentos de inyección [Baraza00, Baraza02] fue el estudio de la respuesta del sistema microcomputador tolerante a fallos en presencia de fallos transitorios y permanentes. Los parámetros utilizados durante los distintos experimentos fueron:

1. **Técnica de inyección**: Órdenes del simulador.
2. **Número de fallos**: n = 3000 fallos por campaña.
3. **Carga de trabajo**: Serie aritmética de *j* números enteros (según se puede ver en la fórmula 1).
4. **Tipos de fallos**: Se han inyectado fallos transitorios y permanentes de los siguientes tipos:
    - Transitorios: *Bit-flip*, *Stuck-at* (‘0’, ‘1’), Indeterminación y Retardo (*Delay*).
    - Permanentes: *Stuck-at* (‘0’, ‘1’), Indeterminación, Retardo (*Delay*) y *Open-line* (Alta impedancia).

    Como se puede observar, se han aumentado los modelos de fallos que se inyectaron respecto a trabajos anteriores.
5. **Lugar de la inyección**: Los fallos han sido sistemáticamente inyectados en todas las señales y variables del modelo. Los fallos no se inyectaron en la CPU de repuesto ya que ésta está desconectada mientras el sistema está trabajando adecuadamente (en inglés *Cold Stand–by Sparing*).
6. **Instante de inyección**: Distribuido uniformemente en el rango $[0, t_{Workload}]$, donde $t_{Workload}$ es la duración de la carga de trabajo sin fallos.
7. **Duración de la simulación**: El tiempo total de simulación incluye el tiempo de ejecución de la carga de trabajo más el tiempo de recuperación de la CPU de repuesto ($t_{Simulación} = t_{Workload} + t_{CPU\text{-}Repuesto}$).
8. **Duración de los fallos**: Se han inyectado fallos cortos, con una duración igual a una fracción del período del reloj (los más comunes según [Cha93]), así como fallos más largos para asegurarnos la activación de los mismos. Se han considerado tres casos:

a) Fallos transitorios con una duración generada aleatoriamente en el rango [0.1T – 10.0T], donde T es el ciclo de reloj de la CPU,

b) Fallos transitorios con una duración fija de 100T, y

c) Fallos permanentes.

9. **Resultados de los análisis**: En este caso, se ha analizado la patología de los errores, midiendo las latencias de propagación, detección y recuperación y calculando las coberturas de detección y recuperación, rellenando el *grafo de predicados de los mecanismos de tolerancia a fallos*, el cual se muestra en la Figura 28. Este grafo representa el proceso seguido por los fallos desde que son inyectados en el sistema hasta su eventual detección y recuperación por los mecanismos de tolerancia a fallos [Arlat93]. En caso contrario, se produce una avería.

Además, se han obtenido diferentes parámetros:

- Porcentaje de errores activados, P_A.

$$P_A = \frac{N_{Activados}}{N_{Inyectados}} = \frac{N_{Activados}}{n} \qquad (2)$$

donde $N_{Activados}$ representa el número de errores activados. Definimos un error activado como un error que se ha originado por un fallo activado (un fallo que produce un cambio en cualquier señal o variable del modelo) y es propagado a las señales de propagación, que en este caso son las señales de la arquitectura estructural externa del sistema.

- Cobertura de detección de errores. Se han calculado dos tipos de estimadores:
  - ⇨ Cobertura de detección de los mecanismos:

$$C_{d(mecanismos)} = \frac{N_{Detectados}}{N_{Activados}} \qquad (3)$$

siendo $N_{Detectados}$ el número de errores detectados por los diferentes mecanismos de detección del sistema.

  - ⇨ Cobertura global de detección del sistema:

$$C_{d(sistema)} = \frac{N_{Detectados} + N_{No-efectivos}}{N_{Activados}} \qquad (4)$$

Los errores que no afectan al resultado de la aplicación se denominan errores no efectivos o $N_{No\text{-}efectivos}$. Este tipo de errores se produce normalmente cuando el fallo es sobrescrito o permanece latente en una parte no usada del sistema. En este último caso, el fallo podría activarse si dicha parte fuera sensibilizada por el algoritmo ejecutado en el sistema. Los errores no efectivos se relacionan con la redundancia intrínseca del sistema. Es por esta razón por la que se define una cobertura más global, denominada cobertura global de detección del sistema.

- Cobertura de recuperación de errores, dividida también en dos tipos:
  - ⇨ Cobertura de recuperación de los mecanismos:

$$C_{r(mecanismos)} = \frac{N_{Detectados\_recuperados}}{N_{Activados}} \qquad (5)$$

siendo $N_{Detectados\_recuperados}$ el número de errores recuperados por los diferentes mecanismos de recuperación del sistema, después de haber sido detectados por los mecanismos de detección del mismo.

⇨ Cobertura global de recuperación del sistema:

$$C_{r(sistema)} = \frac{N_{Detectados\_recuperados} + N_{No-efectivos}}{N_{Activados}} \qquad (\mathbf{6})$$

donde $N_{No\text{-}efectivos}$ tiene el mismo significado que en la fórmula 4.

- Latencias de propagación, detección y recuperación:
  - ⇨ Latencia de propagación (denominada *dormancy* en [Laprie92]): $L_p = t_p - t_{inj}$, donde t_p es el instante de tiempo en el error se ha propagado, mientras que t_{inj} es el instante de inyección.
  - ⇨ Latencia de detección: $L_d = t_d - t_p$, donde t_d es el instante de tiempo en el que el error es detectado.
  - ⇨ Latencia de recuperación: $L_r = t_r - t_d$, donde t_r es el instante de tiempo en el que los mecanismos de recuperación acaban con el proceso de recuperación del error.

Los resultados obtenidos se han agrupado teniendo en cuenta dos aspectos de la validación de los sistemas tolerantes a fallos: la influencia de la duración del fallo y la contribución de los diferentes mecanismos de detección y recuperación. La Figura 33, la Figura 34 y la Figura 35 muestran los *grafos de predicados* para cada duración de los fallos inyectados. Estas figuras muestran una información muy exhaustiva sobre la patología de los fallos y de los errores del sistema y la evolución de los mismos.

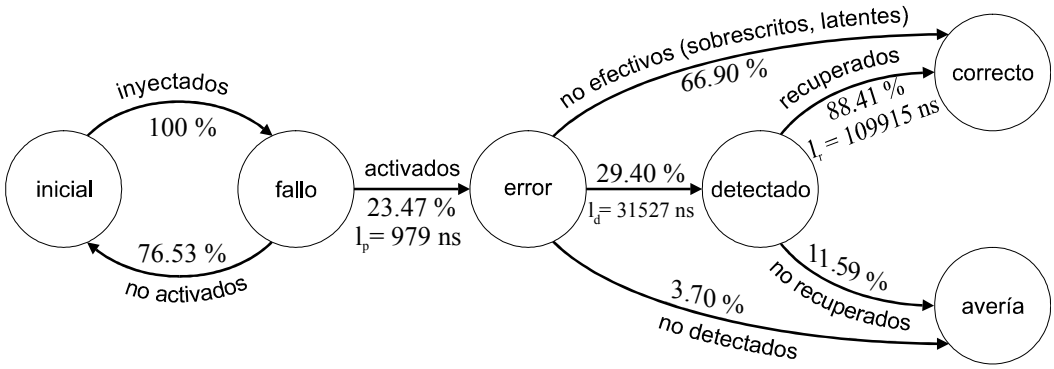


Figura 33. Grafo de predicados de los mecanismos de tolerancia a fallos [Baraza02]. Fallos Transitorios. Duración = Uniforme [0.1T-10.0T].

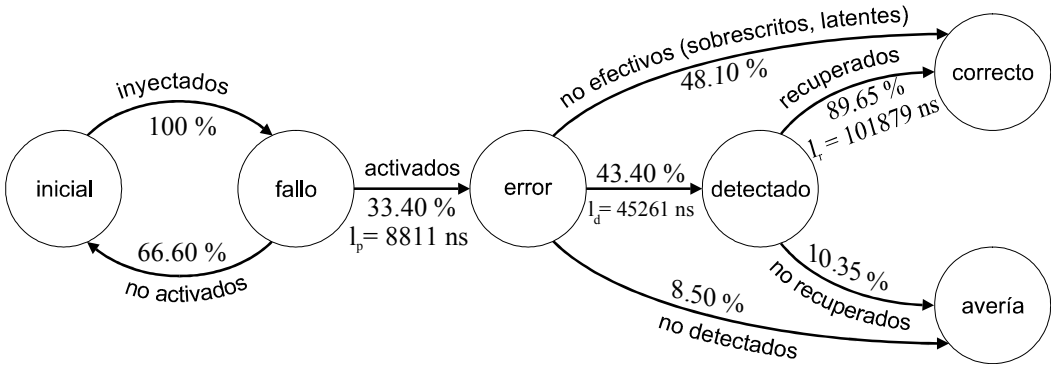


Figura 34. Grafo de predicados de los mecanismos de tolerancia a fallos [Baraza02]. Fallos Transitorios. Duración = 100T.

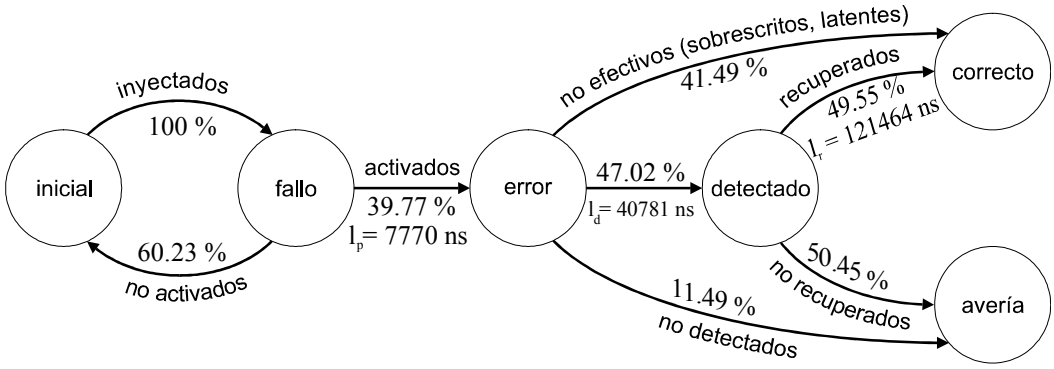


Figura 35. Grafo de predicados de los mecanismos de tolerancia a fallos [Baraza02]. Fallos Permanentes.

La Tabla 17 muestra un resumen de los principales resultados obtenidos en estos trabajos. Se puede observar que a medida que la duración de los fallos decrece:

- P_A decrece. Los fallos más cortos tienen una influencia menor en el funcionamiento del sistema.
- $C_{d(mecanismos)}$ decrece. Los fallos con una duración menor son más difíciles de detectar y recuperar. Sin embargo, $C_{d(sistema)}$ sigue la tendencia contraria. Esto es debido al crecimiento en el porcentaje de errores no efectivos, como se puede ver en la Figura 33, la Figura 34 y la Figura 35.
- C_r sigue el mismo comportamiento que C_d para los fallos transitorios. Esto significa que casi todos los errores detectados son recuperados. Sin embargo, C_r decrece notablemente para los fallos permanentes. La razón puede ser que una porción de los fallos permanentes afecta a la CPU de repuesto, incluso si estos fallos no se han generado en esta CPU.
- $L_p < L_d << L_r$. A pesar de que los valores de las latencias medias no parecen tener una dependencia clara respecto la duración del fallo, sí que siguen la misma tendencia que otros trabajos anteriores, como por ejemplo [DGil99b].

	Duración de los fallos		
Parámetros	**[0.1T-10.0T]**	**100T**	**Permanentes**
P_A (%)	23.47	33.40	39.77
$C_{d\ (mecanismos)}$ (%)	29.40	43.41	47.02
$C_{d\ (sistema)}$ (%)	96.31	91.52	88.52
$C_{r\ (mecanismos)}$ (%)	24.29	35.13	18.36
$C_{r\ (sistema)}$ (%)	91.19	83.23	59.85
Lp (ns)	979	8811	7770
Ld (ns)	31527	45261	40781
Lr (ns)	109915	101879	121464

Tabla 17. Porcentaje de errores activados, coberturas y latencias según la duración del fallo [Baraza02].

Las siguientes tablas (Tabla 18 y Tabla 19) muestran el porcentaje de errores detectados y recuperados así como las latencias medias obtenidas por los distintos mecanismos de detección y recuperación del modelo. Las principales conclusiones que se pueden obtener son:

- Respecto a la detección, la paridad detecta un mayor porcentaje de errores que el Temporizador de Guardia (WDT en la tabla) para fallos cuya duración está en el rango

[0.1T, 10.0T], mientras que para duraciones mayores (100T y permanentes), el método más efectivo en la detección de errores es el Temporizador de Guardia.

- L_d (Paridad) << L_d (WDT). La Paridad detecta los errores antes que el temporizador de guardia.
- En cuanto a la recuperación, los fallos permanentes provocan un mayor porcentaje de errores permanentes, que se traduce en la activación de la CPU de repuesto.
- Para fallos en el rango [0.1T-10.0T], % CPU Repuesto > % *Back–off*[26] >> % *Checkpoint*[27], mientras que para fallos iguales a 100T, % CPU Repuesto > % *Checkpoint* >> % *Back–off*.
- En el rango [0.1T-10.0T], L_r (*Back–off*) < L_r (*Checkpoint*) << L_r (CPU Repuesto), mientras que para fallos con duración 100T y permanentes, L_r (*Checkpoint*) < L_r (*Back–off*) < L_r (CPU Repuesto).

Mecanismo Detección	**% Errores Detectados**			**Latencia Media (ns)**		
	[0.1T-10.0T]	**100T**	**Perm.**	**[0.1T-10.0T]**	**100T**	**Perm.**
Paridad	61.35	38.85	42.42	3021	6062	6210
WDT	38.65	61.15	57.58	76779	70165	66255

Tabla 18. Porcentaje de errores detectados y latencia de detección media según los mecanismos de detección [Baraza02].

Mecanismos Recuperación	**% Errores Recuperados**			**Latencia Media (ns)**		
	[0.1T-10.0T]	**100T**	**Perm.**	**[0.1T-10.0T]**	**100T**	**Perm.**
Back–off	22.81	4.26	6.39	13916	81333	54810
Checkpoint	5.85	44.32	1.83	35932	46577	25194
CPU Repuesto	71.34	51.42	91.78	146667	151244	128022

Tabla 19. Porcentaje de errores recuperados y latencia de recuperación media según los mecanismos de recuperación [Baraza02].

Para finalizar algunas conclusiones obtenidas han sido:

- Se han verificado las buenas prestaciones de la técnica de inyección (manipulación automática de las señales y variables del modelo).
- Se han presentado nuevos modelos de fallos que mejoran los utilizados en trabajos previos. Estos nuevos modelos de fallos son posibles gracias a las capacidades de programación que presenta el simulador de VHDL utilizado [Model98, Model01a].
- Respecto a los resultados de la validación del sistema microcomputador tolerante a fallos, podemos concluir que:
  - ⇨ A medida que la duración de los fallos crece, el porcentaje de fallos activados crece, la cobertura de los mecanismos aumenta significativamente y la cobertura del sistema decrece ligeramente.
  - ⇨ Los fallos permanentes provocan una gran caída en las coberturas de recuperación.
  - ⇨ No se ha observado una dependencia clara de las latencias y la duración de los fallos. Al igual que en trabajos anteriores, se ha observado $L_p < L_d << L_r$.

[26] *Back–off cycle*: ejecución de la instrucción anterior.
[27] *Checkpointing*: puntos de recuperación.

- ⇨ Para fallos transitorios de corta duración (los más difíciles de detectar y recuperar), las coberturas globales de detección y recuperación han sido muy altas (96 % y 91 % respectivamente).
- ⇨ La paridad es el mecanismo más efectivo a la hora de detectar fallos transitorios cortos. Para fallos largos (100T y permanentes), es el temporizador de guardia el mecanismo de detección más efectivo. La paridad tiene una latencia media menor que el temporizador de guardia, lo que significa que detecta antes los errores.
- ⇨ Los mecanismos de recuperación más efectivos para fallos cortos han sido el *Back–off cycle* y la CPU de repuesto. Para fallos largos, han sido el *Checkpoint* y la CPU de repuesto. Las latencias más altas siempre se corresponden con la CPU de repuesto.

En [DGil00, Gracia02c] se introdujeron algunas variantes sobre la validación del sistema microcomputador tolerante a fallos. El cambio principal fue la utilización de dos cargas de trabajo distintas durante la validación. Es decir, se realizaron una serie de experimentos de inyección de fallos para validar la Confiabilidad del sistema tolerante a fallos anterior. Estos nuevos experimentos de inyección de fallos presentan los mismos parámetros que los experimentos recientemente explicados, excepto que ahora se utilizan dos cargas de trabajo y varían tanto los tipos como la duración de los fallos que se inyectaron. Es decir, las diferencias con el experimento anterior fueron:

- **Carga de trabajo**: Se han utilizado dos cargas de trabajo:
  a. Serie aritmética de *j* números enteros (ver fórmula 1).
  b. Algortimo de ordenación *bubblesort*, aplicado a *n* números enteros, siendo n = 6.
- **Tipos de fallos**: Transitorios, del tipo *Bit-flip*, *Stuck-at* (‘0’, ‘1’), Indeterminación y Retardo (*Delay*).
- **Duración de los fallos**: Se han generado aleatoriamente en los rangos [0.1T-10.0T] y [0.01T-1.0T], donde T es el ciclo de reloj de la CPU. Se han inyectado fallos cortos, con una duración igual a una fracción del período del reloj, así como fallos más largos.

Para la obtención de resultados se ha utilizado la misma metodología que en el experimento anterior. Es decir, se ha analizado la patología de los errores, midiendo las latencias de propagación, detección y recuperación y calculando las coberturas de detección y recuperación, rellenando de esta manera el *grafo de predicados* que se muestra en la Figura 28, además de calcular los mismos parámetros expuestos anteriormente en las fórmulas 2 – 6.

Los resultados se muestran en la Tabla 20, donde se pueden ver los porcentajes de errores activados, coberturas y latencias respecto de la duración de los fallos para ambas cargas de trabajo, mientras que la Figura 36 muestra detalladamente la patología de los fallos y errores para ambas duraciones de los fallos inyectados. Como se puede observar, se observa la misma tendencia que en el trabajo anterior con respecto a los fallos propagados, coberturas y latencias. Como se puede ver en los resultados de la Tabla 20, éstos reproducen la influencia global de la duración del fallo que se muestra en [DGil99b], observándose además que aunque este comportamiento es independiente de la carga de trabajo, los valores de las coberturas y latencias para el *bubblesort* son usualmente un poco mayores.

	Serie Aritmética		*Bubblesort*	
Parámetros	**[0.01T-1.0T]**	**[0.1T-10.0T]**	**[0.01T-1.0T]**	**[0.1T-10.0T]**
P_A (%)	20.07	23.47	20.06	25.07
$C_{d\,(mec)}$ (%)	25.42	29.40	27.20	30.24
$C_{d\,(sys)}$ (%)	98.51	96.31	97.58	97.34
$C_{r\,(mec)}$ (%)	20.60	24.29	23.79	26.00
$C_{r\,(sys)}$ (%)	93.69	91.19	94.18	93.09
Lp (ns)	973	979	1550	1824
Ld (ns)	35524	31527	38594	33787
Lr (ns)	89554	109915	114136	123976

Tabla 20. Porcentaje de errores activados, coberturas y latencias según la duración de los fallos [DGil00].

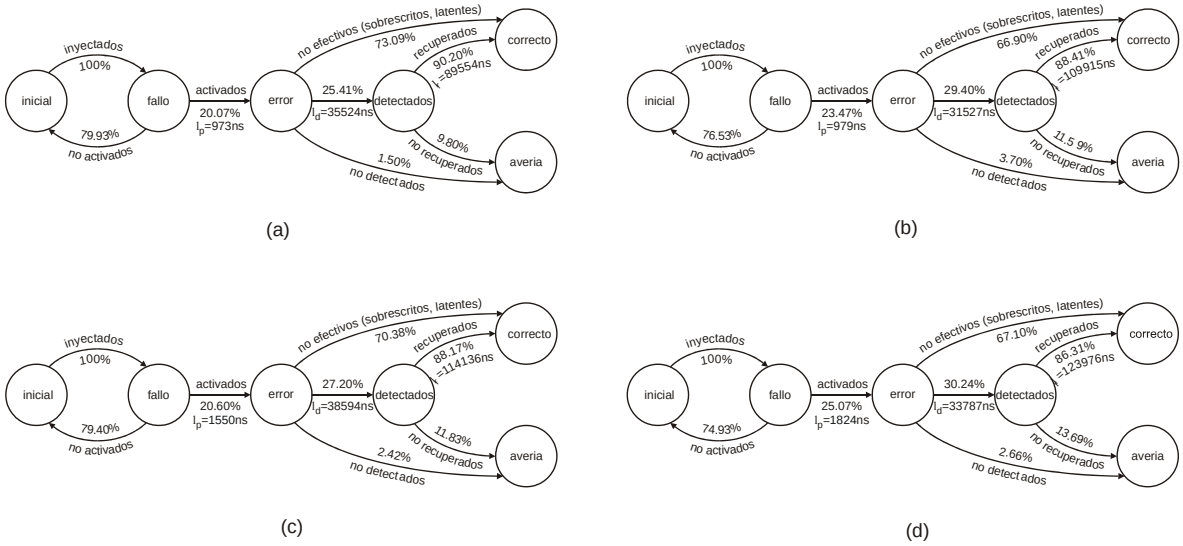


Figura 36. Grafo de predicados de los mecanismos de tolerancia a fallos [DGil00]. Para la serie aritmética (a) duración = uniforme [0.01T-1.0T] (b) duración = uniforme [0.1T-10.0T]. Para el *Bubblesort* (c) duración = uniforme [0.01T-1.0T] (d) duración = uniforme [0.1T-10.0T]

La Tabla 21 muestra separadamente la contribución de los diferentes mecanismos de detección y recuperación del sistema en los valores de las coberturas y latencias. Como se puede ver, estos valores siguen la misma tendencia que el trabajo presentado anteriormente en [Baraza02].

	Carga de trabajo = Serie Aritmética				Carga de trabajo = ***Bubblesort***			
Mecanismos Detección	**% Errores Detectados**		**Latencia media (ns)**		**% Errores Detectados**		**Latencia media (ns)**	
	[0.1T-10.0T]	**[0.01T-1.0T]**	**[0.1T-10.0T]**	**[0.01T-1.0T]**	**[0.1T-10.0T]**	**[0.01T-1.0T]**	**[0.1T-10.0T]**	**[0.01T-1.0T]**
Paridad	61.35	56.21	3021	4077	62.60	53.69	7694	9687
WDT	38.65	43.79	76779	75888	37.40	46.31	77377	72756
Mecanismos Recuperación	**% Errores Recuperados**		**Latencia media (ns)**		**% Errores Recuperados**		**Latencia media (ns)**	
	[0.1T-10.0T]	**[0.01T-1.0T]**	**[0.1T-10.0T]**	**[0.01T-1.0T]**	**[0.1T-10.0T]**	**[0.01T-1.0T]**	**[0.1T-10.0T]**	**[0.01T-1.0T]**
Back-off	22.81	34.68	13916	23472	23.72	25.25	98360	93556
Checkpoint	5.85	7.26	35932	13180	6.65	8.29	15996	19726
CPU Repuesto	71.34	58.06	146667	13866	69.63	66.46	141346	133467

Tabla 21. Contribución de los mecanismos de detección y recuperación a los resultados de coberturas y latencias [DGil00].

Uno de los aspectos más importantes de estos trabajos es el aumento de los modelos de fallos que se pueden inyectar con respecto a trabajos anteriores. La Tabla 22 refleja la influencia de los modelos de fallos transitorios en los parámetros de la Confiabilidad mediante la comparación de dos conjuntos de modelos de fallos:

a. Conjunto de fallos 1: *stuck-at* (‘0’, ‘1’), *bit-flip*, indeterminación y retardo (*delay*). Estos son los modelos de fallos utilizados en [DGil00].

b. Conjunto de fallos 2: *stuck-at* (‘0’, ‘1’), indeterminación y *open-line* (alta impedancia). Este conjunto de fallos se utilizó en [DGil99b].

En esta tabla podemos ver que el *conjunto de fallos 1* presenta un mayor valor en el porcentaje de fallos activados (P_A), y unos valores inferiores para las coberturas de detección y recuperación de los mecanismos. Esto puede ser debido al mayor impacto del nuevo conjunto de fallos, el cual incluye más tipos de fallos transitorios.

Parámetros	**Modelo de Fallos 1**	**Modelo de Fallos 2**
P_A (%)	23.47	22.83
$C_{d\,(mec)}$ (%)	29.40	37.52
$C_{d\,(sys)}$ (%)	96.31	96.35
$C_{r\,(mec)}$ (%)	24.29	30.80
$C_{r\,(sys)}$ (%)	91.19	89.64
Lp (ns)	979	1766
Ld (ns)	31527	31419
Lr (ns)	109915	114267

Tabla 22. Influencia de los modelos de fallos transitorios en parámetros de la Confiabilidad [DGil00]. Duración = Uniforme [0.1T – 10.0T]. Carga de trabajo = Serie aritmética.

En general, las principales conclusiones que se han obtenido en [DGil00] han sido:

- Se han verificado las buenas prestaciones de la técnica de inyección (manipulación automática de las señales y variables del modelo), especialmente la facilidad de implementación y la alta controlabilidad y observabilidad de los experimentos de inyección (como se ha podido ver también en [Baraza00, Baraza02]).

- Para fallos transitorios de corta duración ([0.01T-1.0T]), la cobertura global de detección y recuperación del sistema es bastante alta: sobre el 98 % y el 94 %, respectivamente.
- El mecanismo de detección más efectivo ha sido la paridad (67 % de errores detectados), teniendo además una menor latencia de detección.
- El mecanismo de recuperación más efectivo ha sido la CPU de repuesto, seguido del *Back–off cycle* y el *Checkpointing*. Las mayores latencias de recuperación se corresponden con la CPU de repuesto y los puntos de recuperación (*Checkpoint*). Es decir, la CPU de repuesto es el mecanismo que más errores recupera, pero los recupera más lentamente.
- La dependencia general de la duración del fallo, así como la contribución relativa de los diferentes mecanismos de detección y recuperación ha sido verificada mediante el uso de dos cargas de trabajo, no habiéndose encontrando grandes diferencias.
- Los modelos de fallos utilizados tienen una incidencia mayor en el funcionamiento normal del sistema respecto a modelos de fallos anteriores (p. ej. el modelo de fallos en [DGil99b]). Este hecho se ha reflejado en el aumento del porcentaje de fallos activados y en la disminución de las coberturas de los mecanismos.
- En general, estos resultados reproducen la tendencia de la influencia de la duración de los fallos mostrada en [DGil99b].
- La información presentada en este trabajo puede ser utilizada para mejorar el diseño de los mecanismos de detección y recuperación así como para optimizar los valores de las coberturas y las latencias.

Así pues, una vez que se ha establecido la viabilidad de las órdenes del simulador, y la funcionalidad del modelo en VHDL de un sistema microprocesador tolerante a fallos, el siguiente paso fue la aplicación de las otras dos técnicas de inyección (perturbadores y mutantes) a nuestro sistema. El primer intentó se realizó en [Gracia00a]. En este trabajo se mostraban los primeros resultados de la validación utilizando las tres técnicas, los cuales sirvieron de introducción al trabajo desarrollado posteriormente en [Gracia01a, Gracia01b, DGil03a].

Por lo tanto, en dichos trabajos se estudian y comparan las tres técnicas de inyección explicadas anteriormente (órdenes del simulador, perturbadores y mutantes) mediante diferentes experimentos de inyección en el sistema microprocesador tolerante a fallos utilizado en trabajos anteriores [DGil99b, Baraza00, Baraza02, Gracia02c].

Durante los experimentos de inyección llevados a cabo en estos trabajos se inyectaron fallos transitorios y permanentes durante la ejecución de dos cargas de trabajo diferentes. Se ha estudiado la patología de los errores propagados y se han medido sus latencias y calculado las diferentes coberturas, según se pudo ver en las fórmulas 2 – 6. Concretamente, los parámetros de los experimentos de inyección fueron:

1. **Técnica de inyección**: Órdenes del simulador, perturbadores y mutantes.
2. **Número de fallos**: n = 3000 fallos por campaña.
3. **Carga de trabajo**:
   - Serie aritmética de *j* números enteros (según la fórmula 1).
   - Algoritmo de ordenación *bubblesort* para *n* números (*n* = 6).
4. **Tipos de fallos**: Transitorios y permanentes. Los modelos de fallos inyectados con cada técnica se pueden ver en la Tabla 15. Como se puede comprobar, en estos experimentos se van a inyectar un gran número de modelos de fallos, incluyendo nuevos modelos no utilizados anteriormente, como son el *Pulse*, *Short*, *Bridge*, *Stuck-open*, etc. Una descripción más detallada de los mismos se puede estudiar en el capítulo correspondiente (Capítulo 1, punto 4.3 “Modelos de fallos”).

5. **Lugar de la inyección**: Dependiendo de la técnica utilizada, los fallos se inyectan en diferentes lugares. Utilizando la técnica de las órdenes del simulador, los fallos han sido sistemáticamente inyectados en cualquier señal y variable del modelo. Con la técnica de los perturbadores, los fallos se han inyectado en las señales de la arquitectura estructural externa y utilizando la técnica de los mutantes, los fallos se han inyectado en los elementos sintácticos de la arquitectura comportamental de los componentes. En cualquier caso, los fallos no se inyectaron en la CPU de repuesto por las razones antes comentadas.
6. **Instante de inyección**: Distribuido uniformemente en el rango [0, $t_{Workload}$], donde $t_{Workload}$ es la duración de la carga de trabajo sin fallos.
7. **Duración de la simulación**: El tiempo total de simulación incluye el tiempo de ejecución de la carga de trabajo más el tiempo de recuperación de la CPU de repuesto ($t_{Simulación} = t_{Workload} + t_{CPU\text{-}Repuesto}$)..
8. **Duración de los fallos**: Para los fallos transitorios, se han generado aleatoriamente en el rango [0.1T-10.0T], donde T es el ciclo de reloj de la CPU. Se han inyectado fallos cortos, con una duración igual a una fracción del período del reloj, que son los más difíciles de detectar.
9. **Resultados de los análisis**: Como en los casos anteriores, se ha analizado la patología de los errores, midiendo las latencias de propagación, detección y recuperación y calculando las coberturas de detección y recuperación. Como resultado del análisis, se ha podido completar el *grafo de predicados de los mecanismos de tolerancia a fallos* (según se puede ver en la Figura 28), así como los parámetros de las fórmulas 2 – 6.

A continuación se muestran los resultados obtenidos a partir de estas condiciones previas. De la Figura 37 a la Figura 42 se muestran los porcentajes de errores activados, las coberturas y las latencias medias en función de la duración de los fallos (transitorios o permanentes), la técnica de inyección y la carga de trabajo (serie aritmética y *bubblesort*).

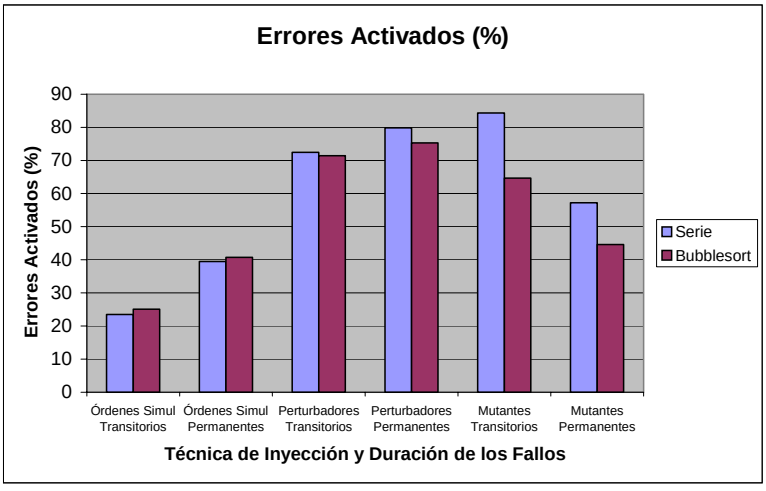


Figura 37. Porcentaje de errores activados respecto de la técnica de inyección y la duración de los fallos [DGil03a].

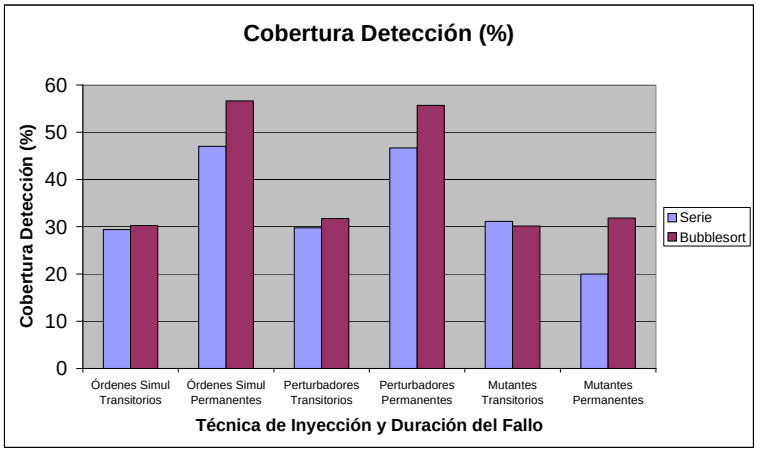


Figura 38. Cobertura de detección respecto de la técnica de inyección y la duración de los fallos [Gracia01b].

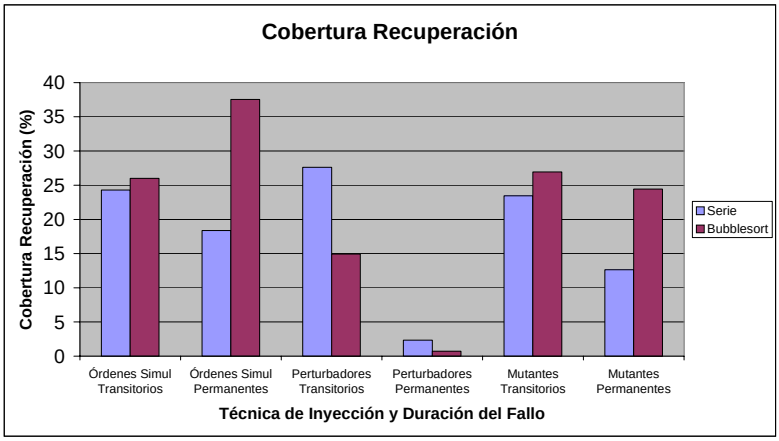


Figura 39. Cobertura de recuperación respecto de la técnica de inyección y la duración de los fallos [Gracia01b].

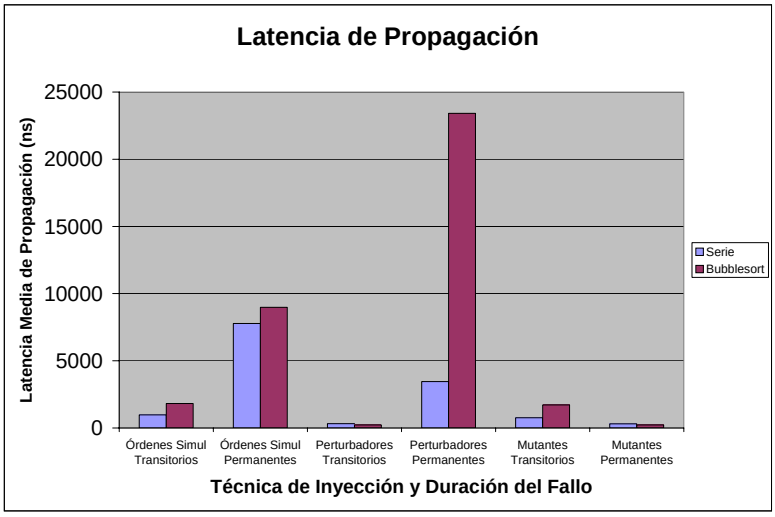


Figura 40. Latencia media de propagación respecto de la técnica de inyección y la duración de los fallos [DGil03a].

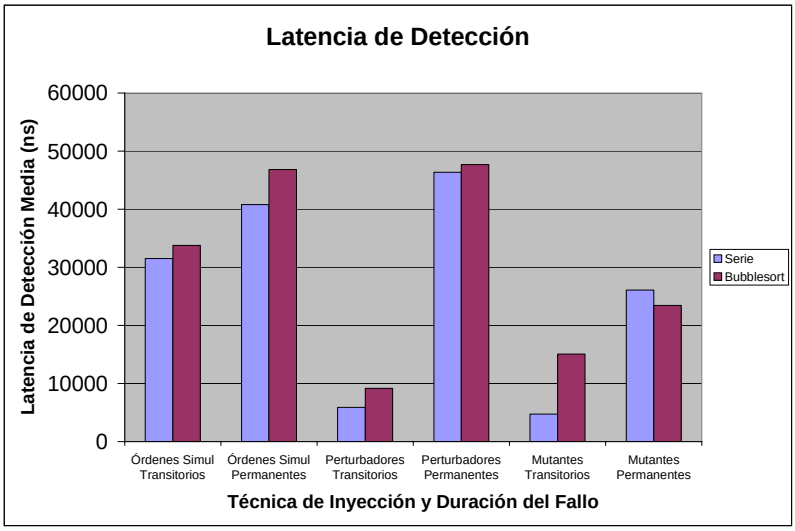


Figura 41. Latencia media de detección respecto de la técnica de inyección y la duración de los fallos [Gracia01b].

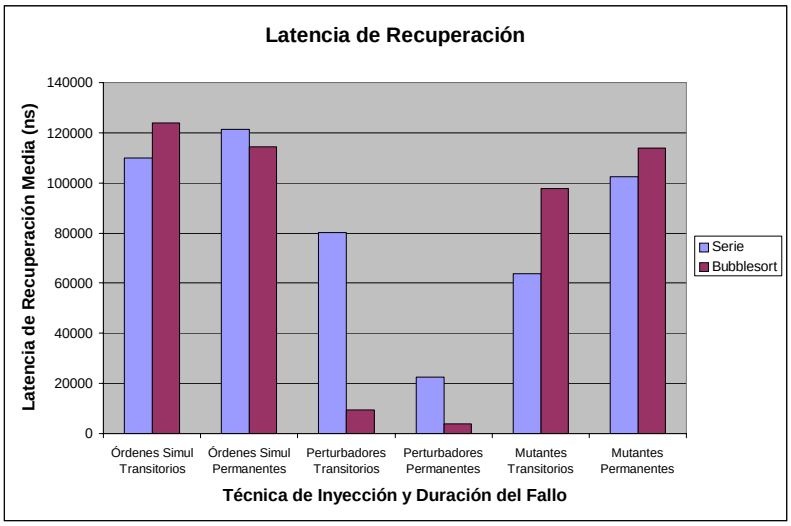


Figura 42. Latencia media de recuperación respecto de la técnica de inyección y la duración de los fallos [Gracia01b].

El análisis de los resultados se ha abordado en función de cinco aspectos:

- La influencia de las técnicas de inyección.
- La influencia de la duración del fallo.
- La influencia de la carga de trabajo.
- La patología de los fallos y errores.
- Los costes de implementación.

Influencia de las técnicas de inyección

- En la Figura 37 podemos observar que la proporción de errores activados (P_A) aumenta notablemente utilizando perturbadores y mutantes. Esto implica que los fallos inyectados utilizando estas dos técnicas han tenido mayor influencia en el funcionamiento del sistema que los fallos inyectados con la técnica de las órdenes del simulador. Esto puede ser debido a la mayor capacidad de modelado de fallos que presentan estas dos técnicas (según se puede ver en la Tabla 15).

- En general, tal y como se aprecia en la Figura 38, no se observa una variación significante en la cobertura de detección (C_d). Sólo utilizando la técnica de los mutantes con fallos permanentes se observa una reducción importante en esta cobertura.
- La Figura 39 no refleja diferencias notables de la cobertura de recuperación (C_r), excepto con la técnica de los perturbadores y fallos permanentes. En este caso, C_r disminuye notablemente respecto de otras técnicas.
- Respecto a las latencias de propagación, la Figura 40 muestra que la latencia de propagación de la técnica de las órdenes del simulador es mayor que la de los perturbadores y mutantes. Esta es la tónica general, excepto en unos pocos casos, como utilizando los perturbadores a la hora de inyectar fallos permanentes con la carga de trabajo del *bubblesort*. En general, esta gráfica indica que los fallos inyectados con las técnicas de los perturbadores y mutantes afectan más rápidamente al sistema, ya que tardan menos en propagarse. Este hecho puede deberse a la localización de la inyección del fallo. Utilizando la técnica de los perturbadores, los fallos se inyectan en las señales de la arquitectura estructural más alta. Como las señales utilizadas para detectar la activación de los fallos también están localizadas en esta arquitectura, el resultado es una latencia de propagación baja. En el caso de los mutantes, los fallos inyectados se modelan mediante cambios sintácticos de los componentes originales, con lo que una vez activado el mutante, el comportamiento de éste provoca las latencias de propagación de la Figura 40.
- La Figura 41 muestra las diferentes latencias de detección (L_d). Para fallos transitorios se cumple que L_d(órdenes del simulador) > L_d(mutantes) > L_d(perturbadores), mientras que para fallos permanentes, la relación que se tiene es L_d(perturbadores) > L_d(órdenes del simulador) > L_d(mutantes). Es decir, para fallos transitorios, mediante la técnica de los perturbadores, los fallos se propagan y se detectan más rápidamente, como se puede ver en la Figura 40 y la Figura 41. En cambio, utilizando la técnica de las órdenes del simulador, los fallos tardan más en propagarse y detectarse. Para fallos permanentes, la tendencia cambia. En este caso, utilizando la técnica de los perturbadores, los fallos tardan más en propagarse y en detectarse, mientras que con la técnica de los mutantes, la propagación y detección se realiza más rápidamente.
- La Figura 42 muestra las diferentes latencias de recuperación (L_r). Tanto para fallos transitorios como para permanentes, se cumple: L_r(órdenes del simulador) > L_r(mutantes) > L_r(perturbadores). Este hecho nos muestra que, independientemente de la duración del fallo, la técnica de los perturbadores recupera antes los errores mientras que la técnica de las órdenes del simulador es la que mayor tiempo emplea para recuperarlos.
- La Figura 40, la Figura 41 y la Figura 42 corroboran los resultados obtenidos en [DGil99b], mostrando $L_p < L_d << L_r$. Como dato adicional, cabe destacar que el mecanismo de los puntos de recuperación (*Checkpointing*) es el mecanismo que más contribuye a la latencia total.
- Las diferencias observadas en P_A, coberturas y latencias se deben a los cambios introducidos en los modelos de fallos y los lugares de inyección. Estos dos factores dependen a su vez de la técnica de inyección.

Influencia de la duración del fallo

En general, los resultados corroboran los obtenidos previamente en [DGil99b, Baraza00 y DGil00]. A medida que la duración de los fallos aumenta:

- La proporción de errores activados (P_A) aumenta excepto en los mutantes. Como es de esperar, los fallos permanentes tienen una mayor influencia en el funcionamiento del sistema.
- La cobertura de detección (C_d) también aumenta, excepto en los mutantes. En este caso, los fallos permanentes son más fáciles de detectar que los fallos transitorios.

- La cobertura de recuperación (C_r) decrece con todas las técnicas. Una porción de los fallos permanentes afecta a las prestaciones de la CPU de repuesto, incluso si los fallos no se han generado en esta CPU de repuesto, afectando de esta manera a la cobertura de recuperación.
- En general, las latencias no reflejan una dependencia clara respecto de la duración del fallo. Solamente la latencia de detección (L_d) aumenta cuando se incrementa la duración del fallo.

Influencia de la carga de trabajo

En este caso, sí que se han encontrado algunas diferencias relevantes:

- La proporción de errores activados (P_A) no refleja diferencias importantes, excepto con los mutantes.
- La cobertura de detección (C_d) muestra diferencias significativas en los fallos permanentes, mientras que la cobertura de recuperación (C_r) muestra importantes diferencias tanto en los fallos transitorios como en los fallos permanentes.
- En cuanto a las latencias, la latencia de propagación (L_p) sólo refleja cambios significativos con los perturbadores con fallos permanentes; la latencia de detección (L_d) presenta diferencias importantes con los mutantes con fallos transitorios mientras que la latencia de recuperación (L_r) presenta suficientes diferencias.

Es evidente que la carga de trabajo afecta notablemente a las medidas de la Confiabilidad, ya que constituye un mecanismo que puede activar los fallos producidos en el *hardware*. Una de las ramas abiertas en esta tesis sería el estudio de la influencia de la carga de trabajo en las prestaciones de los sistemas tolerantes a fallos.

Patología de los fallos y errores

A continuación, la Figura 43, la Figura 44 y la Figura 45 muestran los *grafos de predicados de los mecanismos tolerantes a fallos* para las distintas técnicas, cargas de trabajo y duración de los fallos.

En todos los grafos se puede observar la existencia de un número no despreciable de fallos no recuperados e incluso no detectados. A pesar de que el sistema bajo prueba es un microcomputador tolerante a fallos, hay que puntualizar que es un modelo con propósitos académicos, donde el microcomputador sólo tiene dos mecanismos internos de detección de errores, como son la paridad y el temporizador de guardia. Este hecho condiciona los valores de la cobertura de detección. Como la recuperación comienza después de la detección, la cobertura de recuperación también se ve afectada. Para mejorar las coberturas, sería necesario extender los mecanismos internos de detección de fallos del microcomputador, incluyendo, por ejemplo, excepciones (Memoria no usada, Código de operación incorrecto, Dirección de escritura/lectura no válida, etc.).

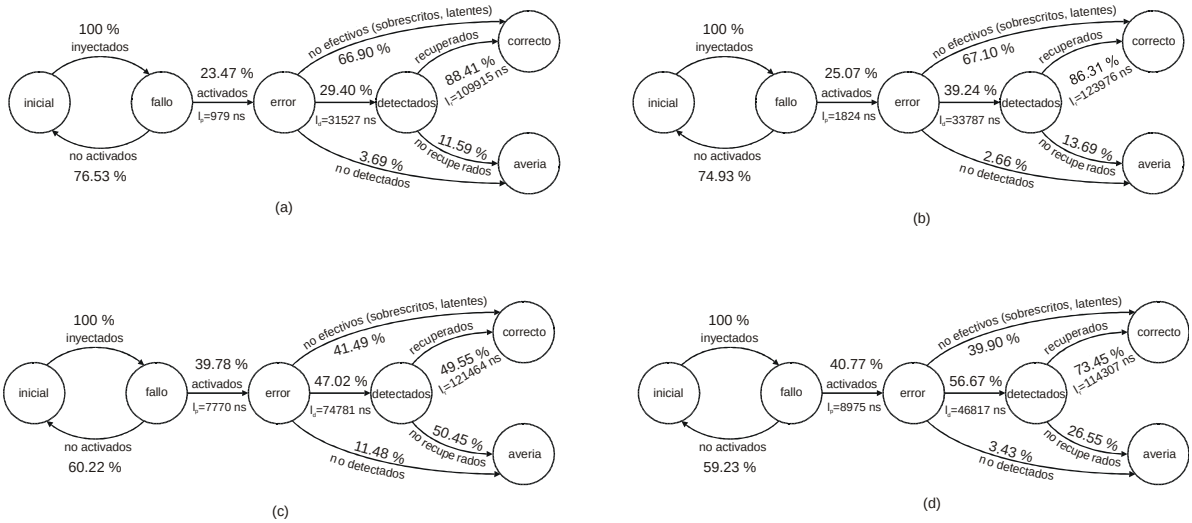


Figura 43. Grafo de predicados de los mecanismos de tolerancia a fallos [DGil03a]. Ordenes del simulador. (a) Fallos Transitorios - Serie Aritmética. (b) Fallos Transitorios – *Bubblesort*. (c) Fallos Permanentes - Serie Aritmética. (b) Fallos Permanentes – *Bubblesort*.

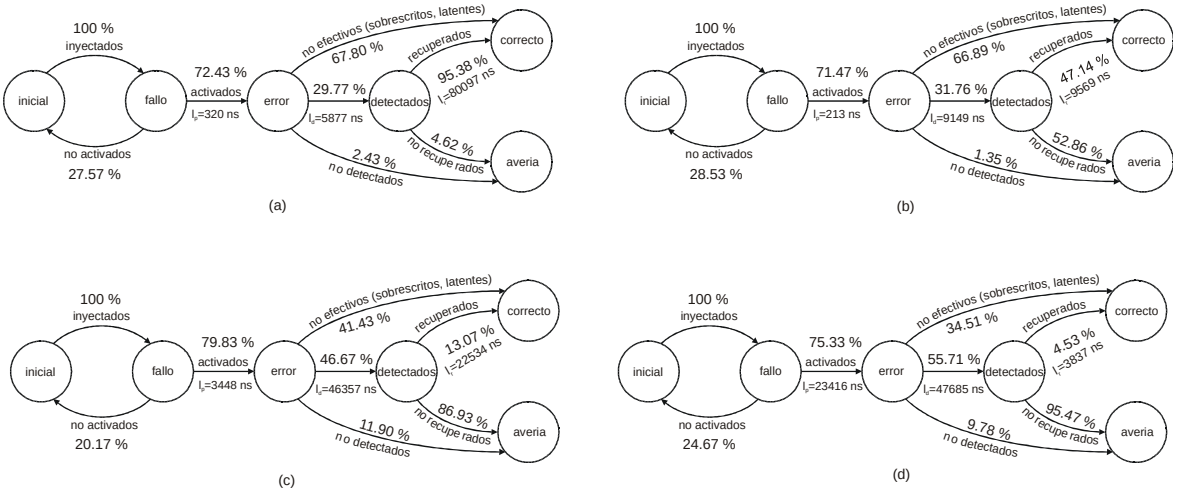


Figura 44. Grafo de predicados de los mecanismos de tolerancia a fallos [DGil03a]. Perturbadores. (a) Fallos Transitorios - Serie Aritmética. (b) Fallos Transitorios – *Bubblesort*. (c) Fallos Permanentes - Serie Aritmética. (b) Fallos Permanentes – *Bubblesort*.

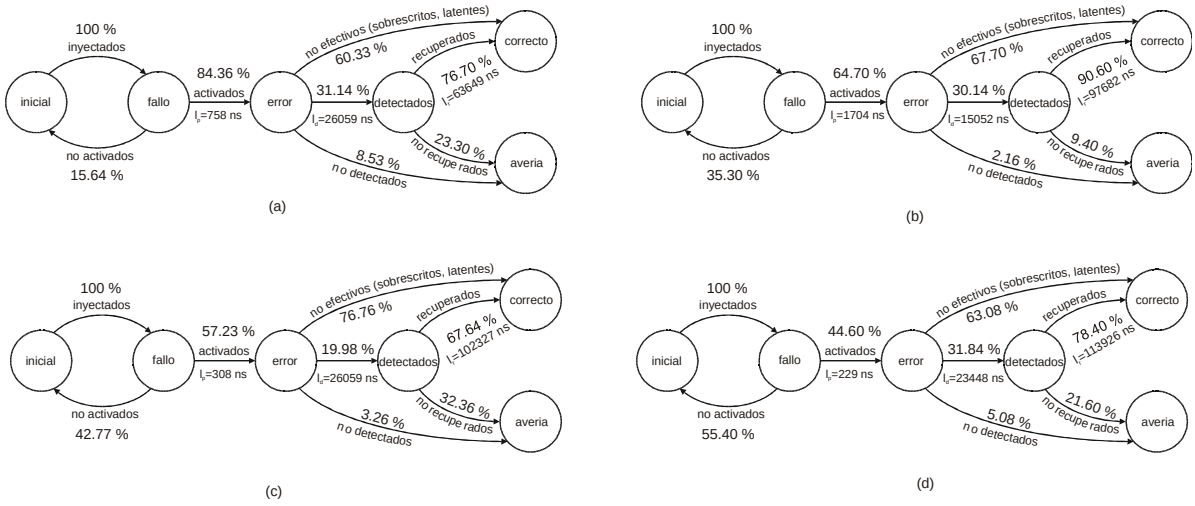


Figura 45. Grafo de predicados de los mecanismos de tolerancia a fallos [DGil03a]. Mutantes. (a) Fallos Transitorios - Serie Aritmética. (b) Fallos Transitorios – *Bubblesort*. (c) Fallos Permanentes - Serie Aritmética. (b) Fallos Permanentes – *Bubblesort*.

Como principales conclusiones de estos grafos (Figura 43 – Figura 44 – Figura 45), podemos decir:

- La proporción de errores activados utilizando la técnica de los mutantes refleja grandes variaciones para las dos cargas de trabajo y las dos duraciones de los fallos. Utilizando la técnica de las órdenes del simulador, este porcentaje depende de la duración del fallo y no de la carga de trabajo. Si se utilizan los perturbadores, este porcentaje es independiente de la carga de trabajo así como de la duración del fallo.
- En cuanto a los fallos detectados, para fallos transitorios los porcentajes son similares, independientemente de la carga de trabajo. Respecto a los fallos permanentes, el porcentaje de fallos detectados es mayor que en los fallos transitorios, y se detectan más fallos cuando se está ejecutando el *bubblesort*. Hay que mencionar sin embargo, dos detalles: i) los porcentajes de fallos permanentes detectados por los mutantes son menores que en los fallos transitorios, ii) las tendencias de los fallos permanentes detectados son iguales, independientemente de la carga de trabajo (% órdenes del simulador ≅ % perturbadores > % mutantes).
- El porcentaje de *fallos activados – no efectivos* en todas las técnicas es casi idéntico e independiente de la carga de trabajo para fallos transitorios, excepto en los mutantes con la serie aritmética que es ligeramente inferior. En cuanto a los fallos permanentes, este porcentaje es similar con los perturbadores y las órdenes del simulador, y es bastante superior con los mutantes, independientemente de la carga de trabajo utilizada.
- Todos los porcentajes de los *fallos activados – no detectados* son similares para todas las técnicas, excepto con la serie para los fallos permanentes. En este caso, el porcentaje aumenta para las técnicas de las órdenes del simulador y los perturbadores.
- En cuanto a los porcentajes de *fallos detectados – recuperados*, éstos presentan grandes divergencias. Quizás los resultados más llamativos son los obtenidos con los perturbadores para fallos permanentes. En estos casos (para ambas cargas de trabajo: serie y *bubblesort*), los porcentajes de *fallos detectados – recuperados* son muy bajos (sobre el 13 % y el 5%, respectivamente). Es decir, con los perturbadores y fallos permanentes se consigue afectar en gran manera a los mecanismos de recuperación de fallos, facilitándose el estudio del efecto de los errores y averías en el sistema.
- En el caso de los porcentajes de *fallos detectados – no recuperados*, éstos son complementarios de los porcentajes de *fallos detectados – recuperados*. Como en el caso anterior, también presentan grandes divergencias, siendo el caso más llamativo el de los

perturbadores con fallos permanentes. En este caso este porcentaje es muy alto, lo que nos indica que se han producido un gran número de averías, tal y como se ha comentado en el punto anterior.

- En cuanto a las latencias, se puede ver:
  - ⇨ En casi todos los casos se observa $L_p < L_d < L_r$, excepto en los perturbadores con fallos permanentes, donde la relación es: $L_p < L_r < L_d$. Hay que mencionar también, que aunque en este caso se recuperan los errores más rápidamente, también es el caso en el que se producen más averías, es decir, que aunque se recuperan más rápido, se recuperan menos errores.
  - ⇨ Las latencias de propagación varían en función de todos los parámetros (técnica de inyección, carga de trabajo y duración de los fallos). Para fallos transitorios, los fallos se propagan más rápidamente con la técnica de los perturbadores que utilizando las otras dos técnicas, siendo en este caso la técnica de las órdenes del simulador la que genera las latencias mayores. En cambio, para fallos permanentes, es la técnica de los mutantes la que provoca la propagación de fallos más rápida.
  - ⇨ Las latencias de detección siguen la misma tónica que las latencias de propagación. Para fallos transitorios, las latencias menores se consiguen con la técnica de los perturbadores y las mayores con las órdenes del simulador, mientras que para fallos permanentes, las latencias más bajas se obtienen con los mutantes, y en este caso, las latencias mayores se generan con los perturbadores.
  - ⇨ Respecto a las latencias de recuperación, para fallos transitorios, las mayores latencias se obtienen mediante las órdenes del simulador. Para fallos permanentes las latencias mayores también se generan con las órdenes del simulador, mientras que las latencias menores se generan con los perturbadores.
  - ⇨ En resumen, se puede decir que para fallos transitorios, las latencias mayores se generan con la técnica de las órdenes del simulador (independientemente de la carga de trabajo), mientras que las latencias más pequeñas se obtienen con los perturbadores (excepto la latencia de recuperación con la serie aritmética). Para fallos permanentes, existe más diversidad en los resultados. Las latencias más bajas se originan con los mutantes, excepto en las latencias de recuperación, que se generan con los perturbadores. Las latencias más altas se producen con los perturbadores o con las órdenes del simulador, dependiendo de la carga de trabajo.

Costes de implementación

Respecto de los costes de implementación, se han corroborado algunas características de las diferentes técnicas de inyección en modelos VHDL que ya se adelantaron en el capítulo 3, "Técnicas de inyección de fallos basadas en VHDL". A partir de los datos obtenidos, podemos asegurar que la técnica de las órdenes del simulador es la más fácil de implementar, ya que la manipulación de señales y variables se realiza directamente, sin requerir ningún cambio en el código.

El principal inconveniente de la técnica de los perturbadores es la inclusión de un cierto número de señales de control en el modelo. Estas señales son utilizadas para activar uno de los perturbadores insertados en el modelo, así como para indicar el tipo de perturbación que se inyectará. Esto añade complejidad tanto a la técnica como al modelo. Sin embargo, aunque la técnica de los perturbadores es más costosa de implementar, también permite una mayor capacidad para modelar fallos que la técnica de las órdenes del simulador, tal y como se puede ver en la Tabla 15.

Respecto a los mutantes, el principal problema, sobretodo cuando se inyectan fallos transitorios, es el elevado coste temporal de la fase de simulación, debido principalmente al cambio dinámico de estado entre la arquitectura con fallo y la arquitectura sin fallo (el cual hay que realizar dos veces). Sin embargo, esta técnica presenta la mayor capacidad de modelado de

fallos de las tres técnicas, ya que puede utilizar toda la capacidad semántica y sintáctica del lenguaje.

Como ejemplo, en la Tabla 23 (extraída de [Gracia01b]) se puede ver el coste temporal de la inyección de 3000 fallos transitorios realizados en un PC con un procesador PII a 350 MHz y 192 Mb de RAM. Como se puede ver, la fase de simulación de la técnica de los mutantes no hace practicable esta técnica.

Técnica de Inyección de Fallos	**Fase de Simulación**		**Fase de Análisis**	
	Serie	*Bubblesort*	Serie	*Bubblesort*
Ordenes del Simulador	2 horas	3 horas	2 horas	2.5 horas
Perturbadores	2.5 horas	6 horas	2.5 horas	5 horas
Mutantes	10 días	14 días	4 horas	5 horas

Tabla 23. Coste temporal de los experimentos de inyección en [Gracia01b].

Respecto al tamaño de los ficheros de traza, los ficheros generados por la técnica de los perturbadores tienen, aproximadamente, seis veces el tamaño de los ficheros producidos con las otras dos técnicas, y para ambas cargas de trabajo. Este incremento es debido a la inclusión de las señales de control en el modelo.

Para finalizar, y como resumen de estos trabajos [Gracia01a, Gracia01b, DGil03a], se puede decir que se han comparado las tres técnicas de inyección de fallos en modelos VHDL: órdenes del simulador, perturbadores y mutantes, y se han implementado algunas extensiones a estas técnicas, como pueden ser los perturbadores bidireccionales, los perturbadores para *buses* y los mutantes transitorios. Se han introducido también un amplio conjunto de modelos de fallos para todas las técnicas, incrementando los modelos de fallos utilizados hasta ahora en la literatura.

Como en los trabajos anteriores, se ha verificado las buenas prestaciones de la técnica de inyección en VHDL y especialmente la alta controlabilidad y observabilidad de los experimentos. En el modelo específico utilizado (microprocesador tolerante a fallos basado en el MARK2), las principales conclusiones que se pueden obtener son:

- Para fallos transitorios, los valores de las coberturas de detección y recuperación muestran pequeñas diferencias. Podemos deducir que con cualquiera de las tres técnicas se puede obtener un valor de las coberturas bastante exacto. Este hecho permite trabajar con deferentes niveles de abstracción del mismo modelo.
- Para fallos permanentes, la técnica de los mutantes muestra algunas discrepancias en los valores de las coberturas de detección y recuperación respecto a las otras técnicas, pudiendo deberse a los modelos de fallos utilizados con esta técnica.
- Los valores medios de las latencias de detección y recuperación para los perturbadores y los mutantes son un poco optimistas (valores más bajos) comparadas con la técnica de las órdenes del simulador. La causa puede estar relacionada con el lugar de la inyección de cada técnica.
- Con relación al coste de implementación, se ha verificado que la técnica de las órdenes del simulador es la más fácil de implementar. Los perturbadores y los mutantes tienen un coste de aplicación y de elaboración mayor debido a los tiempos de recompilación y simulación del sistema. Por otra parte, la técnica de los perturbadores crea unos ficheros de traza mayores que las otras técnicas debido al incremento de las señales de control en el modelo. En cambio, la técnica de los mutantes incrementa el tiempo de simulación debido a la necesidad de guardar y restaurar el estado del sistema cuando se cambia de arquitectura (*sin fallo–con fallo*).

Respecto a la relación existente entre el tipo de modelo del sistema y las técnicas de inyección, se pueden realizar varias recomendaciones. Si se utilizan niveles lógico o RT, las técnicas más adecuadas serían las órdenes del simulador y los perturbadores. A pesar de que la

técnica de los perturbadores es más difícil de implementar, ésta tiene una mayor capacidad de modelado de fallos. Al nivel algorítmico (modelos comportamentales, situación normal en las primeras etapas de diseño), se pueden usar tanto la técnica de las órdenes del simulador como la técnica de los mutantes. Como en el caso anterior, la técnica de los mutantes es más difícil de implementar, pero su capacidad para modelar fallos es mucho más grande. En cualquier caso, el uso combinado de las tres técnicas proporciona un método muy poderoso para validar sistemas modelados en diferentes niveles de abstracción.

5.3 Resumen. Conclusiones y líneas abiertas de investigación

En este capítulo se han presentado una serie de experimentos destinados a probar las tres técnicas de inyección de fallos en VHDL descritas en el capítulo anterior (órdenes del simulador, perturbadores y mutantes). Las principales conclusiones que se pueden obtener de estos experimentos son:

- Se han verificado las buenas prestaciones de las tres técnicas de inyección, así como la alta controlabilidad y observabilidad general de la inyección en VHDL. El uso combinado de las diferentes técnicas de inyección proporciona un método muy poderoso para validar sistemas modelados en diferentes niveles de abstracción.
- Para fallos transitorios, los valores de las coberturas de detección y recuperación muestran pocas diferencias con cualquiera de las tres técnicas. Se puede deducir que las coberturas para este tipo de fallos se pueden obtener con bastante exactitud con cualquiera de las técnicas.
- Respecto a la relación entre el tipo de modelo y las técnicas de inyección, se pueden realizar varias consideraciones. Si se están utilizando niveles lógico o RT (modelos estructurales), es preferible utilizar las técnicas de las órdenes del simulador o los perturbadores. A pesar de que los perturbadores son más difíciles de implementar, presentan una mayor capacidad para modelar fallos. Con modelos comportamentales, (nivel algorítmico) se pueden utilizar las técnicas de los mutantes o las órdenes del simulador. En este caso, son los mutantes los que presentan una mayor capacidad para modelar fallos, ya que pueden usar toda la capacidad semántica y sintáctica del VHDL.
- Se han presentado nuevos modelos de fallos que mejoran los utilizados hasta ahora. Estos nuevos modelos de fallos son posibles gracias a las capacidades de programación que presenta el simulador de VHDL utilizado [Model98, Model01a].
- Los modelos de fallos utilizados tienen una incidencia mayor en el normal funcionamiento del sistema respecto a modelos de fallos utilizados clásicamente en la literatura.
- La información presentada en los distintos experimentos de inyección puede ser utilizada para mejorar el diseño de los mecanismos de detección y recuperación así como para optimizar los valores de las coberturas y las latencias.

Respecto de los costes de implementación de las distintas técnicas, las órdenes del simulador es la más fácil de implementar y la que se ejecuta de una forma más rápida. El tiempo necesario para la inyección del fallo (parar la simulación, modificar la señal o variable, simular la duración del fallo y en el caso de las señales volver a parar la simulación y liberar la señal) es despreciable respecto al tiempo total de simulación. Si además el modelo de sistema a simular es bastante complejo (por ejemplo, los modelos del controlador *Time-Triggered* utilizados en los experimentos explicados en el capítulo 1), se puede considerar nulo el tiempo empleado para la inyección.

En cuanto a la técnica de los perturbadores, ésta tiene un coste de elaboración e implementación mayor que la técnica de las órdenes del simulador, principalmente debido a la implementación de los perturbadores y a la recompilación del modelo (en el cual están incluidos los perturbadores), aunque ésta implementación y recompilación sólo se realiza una vez. Otra desventaja de esta técnica es el aumento del tamaño del fichero de traza al añadir un cierto número de señales de control. Estas señales se utilizan para la activación de uno de los perturbadores y para indicar el tipo de fallo (o perturbación) que se va a inyectar. En cuanto al tiempo de simulación, éste es mayor que el tiempo empleado con la técnica de las órdenes del simulador, debido principalmente a la simulación del modelo más la simulación de los

perturbadores añadidos al mismo. La principal ventaja de esta técnica es la mayor capacidad de modelado de fallos que tiene, tal y como se ha visto anteriormente.

Por último, aunque la técnica de los mutantes es la que ofrece una mayor capacidad para modelar fallos (puede aprovechar toda la capacidad léxica y sintáctica del VHDL), es la que más problemas presenta en su implementación. En primer lugar, es necesario implementar los mutantes y las distintas configuraciones asociadas a los mismos, así como ejecutar su compilación, aunque sólo es necesario realizar este paso una única vez. Otro problema añadido es el aumento en el tiempo de simulación, debido principalmente al cambio de arquitectura (mutada/no mutada) a la hora de inyectar fallos transitorios.

Durante estos experimentos, también se comprobó que los ficheros de traza generados mediante la técnica de los perturbadores eran aproximadamente seis veces más grandes que los obtenidos mediante las técnicas de los perturbadores y los mutantes (para las dos cargas de trabajo). Como se ha comentado antes, esto es debido a la introducción de un cierto número de señales de control en el modelo.

Varias líneas futuras de investigación quedan abiertas en este capítulo. En primer lugar está la problemática de la disminución del tiempo de simulación a la hora de inyectar fallos, principalmente mediante la técnica de los mutantes. Una posible solución se esbozó en el capítulo anterior mediante la utilización de una única arquitectura que incluyera todos los mutantes de la misma.

Otro problema abierto es el estudio de la representatividad de los fallos, en este caso utilizando las técnicas de los perturbadores y de los mutantes. Un primer trabajo realizado sobre representatividad de fallos, aunque en este caso utilizando únicamente la técnica de los comandos del simulador, se puede ver en [Gracia02a].

Por último, otra cuestión pendiente es el estudio de la influencia de la carga de trabajo en los valores de la Confiablilidad, siendo en este caso necesario considerar cargas de trabajo de diferentes tipos.

6 Arquitecturas de bus para sistemas distribuidos de tiempo real tolerantes a fallos. Introducción a la arquitectura Time-Triggered

6.1 Introducción

Con la introducción de sistemas empotrados en aplicaciones críticas, como *fly-by-wire*[28] en aviónica y *drive-by-wire*[29] en automoción, los requerimientos para sistemas empotrados tolerantes a fallos y con una alta fiabilidad crecen. En ese momento, el sistema empotrado pasa a convertirse en un sistema distribuido que incluye mecanismos de sincronización, votadores, control de redundancia, etc.

Uno de los servicios esenciales proporcionados por este tipo de arquitecturas es la transmisión de información desde uno de los componentes del sistema distribuido a otro componente, por lo que para realizar esta comunicación, es necesario tener un medio de comunicación. La topología más usada es la conexión de los diferentes componentes mediante un *bus*, aunque existen otras topologías, como la conexión en estrella, en árbol, etc. De esta forma, el *bus* de comunicaciones se convierte en uno de los componentes principales del sistema, y el protocolo de comunicaciones utilizado en una parte esencial del mismo.

En este capítulo se van a presentar brevemente varias arquitecturas basadas en *bus*, ampliamente utilizadas en aviónica y automoción. El capítulo se cerrará con una amplia descripción de la arquitectura y el protocolo *Time-Triggered*, así como los modelos en VHDL basados en esta arquitectura, los cuales fueron utilizados durante la manufacturación del integrado que implementaba el controlador *TTP/C*. Estos modelos en VHDL también han sido utilizados durante los experimentos de inyección de fallos realizados en el desarrollo de la presente tesis, los cuales se detallarán en el siguiente capítulo.

6.2 Características generales de los sistemas basados en buses

En esta sección se van a estudiar las características generales de los sistemas basados en *buses*, especialmente aquellos sistemas basados en el tiempo (del inglés *Time-Triggered*), ya que son los preferidos generalmente para sistemas integrados que desarrollan funciones críticas. En este punto también se comentarán diferentes sistemas utilizados en la industria aeronáutica y de automoción que presentan estas características [Kopetz97, Sivencrona00, Rushby01a, Rushby01b].

6.2.1 Buses basados en el tiempo (Time-Triggered Buses)

Diferentes sistemas de control utilizados en la industria aeronáutica y de automoción emplean sistemas distribuidos tolerantes a fallos los cuales realizan funciones críticas. Estos sistemas están empezando a construirse de forma modular (están basados en arquitecturas y componentes estándares) e integrada (varios componentes están compartidos por diferentes

[28] Este tipo de aplicaciones implica la ausencia de conexión directa entre el piloto y los mecanismos de control del avión.

[29] En este caso, no existe conexión directa entre el conductor del vehículo y los mecanismos de control del mismo.

funciones, posiblemente con diferentes niveles críticos). Uno de los servicios esenciales proporcionados por este tipo de arquitecturas modulares es el trasvase de información de un componente a otro. Por esta razón, las arquitecturas que se van a comentar en este capítulo se denominan *buses*, ya que uno de los servicios que proporcionan es la comunicación de tipo *multicast* y/o *broadcast*.

Sin embargo, uno de los problemas que surgen al compartir funciones es la propagación de los fallos, los cuales pueden afectar a más de un componente (en contraposición con el funcionamiento anterior, donde cada función estaba aislada del resto, lo que implicaba una gran redundancia de sistemas que encarecían el producto final). Es decir, además de la tolerancia a fallos, dos inconvenientes más surgen de la compartición de funciones: el particionamiento (del inglés *partitioning*, que se puede traducir como la no propagación de una avería producida en una aplicación a otras aplicaciones) y la composición (del inglés *composability*, que implica que una aplicación individual no es afectada por la elección de otras aplicaciones con las que será integrada).

En una arquitectura de *bus* genérica, la aplicación se ejecuta en el *host*[30], al cual están conectados los diferentes sensores y actuadores. Los distintos *hosts* se conectan entre sí a través de un medio de interconexión, utilizando para ello la interfaz adecuada. Las interfaces y la interconexión forman el *bus*, y la combinación de un *host* y su interfaz (o interfaces) se denomina nodo.

Las arquitecturas basadas en *buses* se pueden dividir en dos grandes grupos: basados en el tiempo y basados en eventos (del inglés *event-triggered*). Los *buses* basados en el tiempo realizan sus funciones dependiendo del instante de tiempo en el que se encuentre el sistema (por ejemplo, si han pasado 20 ms desde la recepción del mensaje, entonces, leer el sensor y enviar su valor al resto de nodos del sistema). En cambio, los *buses* basados en eventos realizan sus operaciones en función del evento producido (por ejemplo, si la lectura del sensor cambia, entonces enviar su valor al resto de nodos del sistema).

Uno de los principales problemas de los *buses* basados en eventos, cuando son utilizados en sistemas de control en tiempo real, es su baja predicibilidad a la hora de enviar o recibir mensajes, lo que puede provocar un conflicto por el acceso al *bus*, retardando el envío. De esta manera, es necesario alguna forma de arbitraje en el acceso al mismo, así como asegurar los tiempos de entrega en la transmisión de mensajes cuando hay fallos en el sistema. En este caso, el mayor problema que se puede presentar es la avería del *babbling-idiot*, donde un nodo con fallos transmite constantemente, comprometiendo la operatividad de todo el sistema. Una forma de solucionar estos problemas es con el uso de *buses* basados en el tiempo, donde existe una planificación global estática, en el que cada nodo sabe de antemano cuándo debe enviar o recibir un mensaje. Esta planificación estática permite controlar la avería del *babbling-idiot* mediante un nuevo componente, denominado Guardián del *Bus*, el cual permite transmitir a un nodo únicamente cuando está planificado.

Otra ventaja añadida de los *buses* basados en el tiempo es la reducción del tamaño de los mensajes, ya que al estar predefinido el envío y la recepción del mismo, no es necesario insertar identificador ni del emisor ni del receptor, aumentando de paso el número de mensajes que se pueden enviar (con respecto a los *buses* basados en eventos durante el mismo período de tiempo). Además se reduce una de las fuentes de errores (cambio de la identificación del emisor/receptor debido a un fallo en el *bus*).

Sin embargo, los *buses* basados en el tiempo necesitan una sincronización del reloj tolerante a fallos. Existen dos algoritmos para la sincronización del reloj: basados en la media y basados en eventos. El primero realiza una media entre el reloj local del nodo y el reloj de otro(s) nodo(s) (por ejemplo, comparando el instante de llegada de cada mensaje con el instante de llegada previsto, y haciendo la media), teniendo en cuenta que la sincronización debe ser también tolerante a fallos, es decir, debe omitir o resolver aquellos casos en los que el reloj esté

[30] Puede haber uno o más *hosts*, ejecutando cada uno de ellos una o más aplicaciones.

afectado por un fallo. El segundo método está basado en la capacidad de los nodos de tratar eventos. En este caso, cada nodo envía un mensaje cuando es el momento de realizar la sincronización, y actualiza su reloj cuando ha recibido varios de estos mensajes. Obviamente, esta sincronización también debe ser tolerante a fallos y tratar de forma adecuada aquellos casos en los que el reloj esté afectado por un fallo.

Cada sistema tolerante a fallos se valida en función de unas ciertas *hipótesis de fallos*. Estas hipótesis deben describir los modos de fallos que serán tolerados, así como su número máximo y su tasa de llegada. También deben identificar las diferentes FCU (del inglés *Fault Containment Unit* o Unidad de Contención de Fallo[31]). Un modo de fallo describe el tipo de comportamiento que una FCU con fallo puede exhibir. Sin embargo, se debe asegurar (mediante un diseño muy cuidadoso y un análisis exhaustivo) que las averías en diferentes FCU son independientes entre sí.

Se suelen considerar tres tipos de fallos en este contexto:

- Fallo de valor: se transmite o se recibe un valor erróneo
- Fallo de tiempo: se transmite o se recibe un valor en un instante de tiempo erróneo. Este fallo incluye el fallo de omisión, es decir, cuando ni se transmite ni se recibe el valor.
- Fallo de proximidad espacial: toda la información en una localización física es afectada).

El efecto de estos fallos también puede describirse de tres formas:

- El efecto del fallo puede ser detectado de forma fiable
- El efecto del fallo puede ser simétrico, es decir, el efecto del fallo es el mismo para todos los observadores.
- El efecto del fallo puede ser arbitrario, es decir, el efecto del fallo depende del observador. Este fallo también se denomina fallo bizantino y es el más pernicioso.

6.2.2 *Arquitecturas de* buses *basados en eventos: CAN y ByteFlight*

En este punto se van a comentar dos *buses* basados en eventos ampliamente utilizados en la industria aeronáutica y de automoción.

Implementado por *Bosch* a mediados de los 80, *CAN* (del inglés *Controller Area Network*) es un sistema de comunicaciones (o arquitectura de *bus*) dirigido por eventos [Bosch92]. Para transmitir un mensaje, el canal debe estar sin usar o el mensaje tener la prioridad más alta. Esto provoca que mensajes con baja prioridad puedan ser retardados de forma inaceptable para ciertas aplicaciones (por ejemplo, aplicaciones en tiempo real). El sistema utiliza la técnica CSMA/CA (del inglés *Carrier Sense Multiple Access/Collision Avoidance*) para el acceso al *bus*.

ByteFlight, desarrollado por *BMW* y *Motorola*, *Elmos* e *Infineon*, es un protocolo para aplicaciones críticas de seguridad en automóviles. De manera similar a *CAN*, *ByteFlight* se basa en un proceso orientado a la transmisión de mensajes, es decir, todos los mensajes están a disposición de todos los *hosts* conectados al *bus*. Sin embargo, se diferencia de *CAN* en el acceso al *bus*, el cual está basado en TDMA[32]. La sincronización de cada *host* se realiza mediante un pulso de sincronización que se envía cíclicamente. El intervalo de tiempo entre dos pulsos de sincronización se divide en un cierto número de períodos de tiempo utilizados para la transmisión de los diferentes mensajes (síncronos y asíncronos) [Byteflight01].

---

[31] Una FCU es un componente o un conjunto de componentes que pueden ser afectados por fallos de forma independiente del resto de componentes o FCU.

[32] TDMA: del inglés *Time Division Multiple Access*. Este tipo de acceso se explicará con detalle en el punto 6.3.2.

6.2.3 Arquitecturas de buses basados en el tiempo

En este apartado se van a comentar brevemente una serie de *buses* basados en el tiempo que se utilizan en la industria aeronáutica y de automoción. Se han escogido más ejemplos en este campo porque gran parte del trabajo realizado en esta tesis está basado en la arquitectura *Time-Triggered*, uno de los sistemas más utilizados actualmente por este tipo de industrias.

6.2.3.1 La arquitectura Time-Triggered (TTA)

Desarrollada en la Universidad de Viena [Kopetz94, Kopetz98a, Kopetz98b, Kopetz99] y comercializada actualmente por *TTTech* (http://www.tttech.com), se aplica tanto en la industria automovilística como en aviación (por ejemplo, son clientes de *TTTech* en la industria automovilística *Audi*, *Delphi*, *PSA Peugeot Citroën*, *Renault*, etc. mientras que en la industria de aviación están *Airbus* y *Honeywell*, en trenes *Alcatel*, etc.). Esta arquitectura fue diseñada para su aplicación en sistemas distribuidos de tiempo real tolerantes a fallos.

En esta arquitectura, a las interfaces se las denomina controladores, e implementan el protocolo de comunicaciones, denominado *TTP/C*. Este protocolo proporciona sincronización del reloj, secuenciamiento de mensajes y funciones de transmisión. La hipótesis de fallos incluye fallos arbitrarios y fallos en varios nodos (un fallo por nodo), si ocurre un fallo cada dos rondas. *TTA* adopta la hipótesis de fallos simples: en un nodo, sólo puede fallar el controlador o el Guardián del *Bus*, pero no los dos a la vez. Para sincronizar el reloj utiliza un algoritmo basado en la media. En el punto 6.3 se puede ver una descripción mucho más detallada y exhaustiva sobre esta arquitectura y el protocolo *TTP/C*.

6.2.3.2 La arquitectura SAFEbus

Fue desarrollada por *Honeywell* para servir como parte central del *Airplane Information Management System* (AIMS) del Boeing 777 [Hoyme92, Sweet95]. Las interfaces de conexión al *bus* (denominadas *Bus Interface Units* o BIU) están duplicadas, mientras que el *bus* de comunicaciones está cuadruplicado, solucionando de esta manera los fallos de proximidad espacial. Las BIU ejecutan la sincronización del reloj (realizada mediante un algoritmo basado en eventos), la planificación de los mensajes y el envío de los mismos. Cada BIU actúa como el Guardián del *Bus* del componente vecino y controla su acceso al *bus*. Las hipótesis de fallos de *SAFEbus* incluyen los fallos arbitrarios, fallos en varios nodos (un fallo por nodo) y una alta tasa de ocurrencia de los fallos. *SAFEbus* adopta la hipótesis de fallos simples: como máximo sólo puede fallar un componente de cualquier par.

6.2.3.3 La arquitectura FlexRay

Está siendo desarrollada por un consorcio de diversos fabricantes, los cuales incluyen a *BMW*, *DaimlerChrysler*, *Motorola* y *Philips*. Se diferencia de los anteriores sistemas en que su funcionamiento se basa tanto en el tiempo como en eventos [Berwanger01]. Proporciona también una base global para la sincronización del reloj y transmisión de datos en tiempo real con la latencia del mensaje acotada. Las interfaces, también denominadas controladores de comunicaciones, controlan las líneas de sus interconexiones mediante guardianes de *bus* separados localizados en la interfaz (es decir, con dos *buses* cada nodo tiene tres relojes: uno para el controlador y uno para cada Guardián del *Bus*).

La combinación de eventos y tiempo permite a *FlexRay* dividir cada ciclo de tiempo en una porción estática basada en el tiempo y en una porción dinámica basada en eventos. La comunicación durante esta última parte se realiza mediante el protocolo *ByteFlight* (este protocolo se ha comentado en el punto 6.2.2). Al contrario que en *SAFEbus* y *TTA*, la planificación para la parte basada en tiempo no se carga en los controladores. *FlexRay* divide esta parte en una serie de *slots* (o sub-divisiones de tiempo) de tamaño fijo, y cada controlador y

sus guardianes de *bus* únicamente son informados de aquellos *slots* que le pertenecen para transmitir. Cuando el sistema se pone en marcha, los controladores "aprenden" la planificación.

La hipótesis de fallos comprende fallos asimétricos y fallos espaciales para componentes simples. Para sincronizar el reloj utiliza un algoritmo basado en la media, como *TTA*.

6.2.3.4 La arquitectura Time-Triggered CAN (TTCAN)

Este protocolo es una extensión de *CAN* que presenta una capa de sesión por encima de las capas físicas y de enlace de datos de *CAN*. *TTCAN* proporciona mecanismos que permiten implementar un sistema de comunicación híbrido, admitiendo los dos tipos de comunicaciones: basada en el tiempo y en eventos. En *TTCAN* existen tres clases diferentes de ventanas temporales: *ventanas temporales exclusivas*, *ventanas temporales libres* y *ventanas temporales arbitradas*. Las ventanas temporales exclusivas se asignan a mensajes específicos que se transmiten periódicamente sin competir por el acceso a la red *CAN*. Para soportar la comunicación basada en tiempo, cualquier nodo necesita una base temporal, la cual es proporcionada por un reloj interno o externo. Cualquier mensaje recibido o transmitido almacena el instante de tiempo respecto al punto de referencia del mensaje. Después de completar un mensaje correctamente, este valor almacenado es proporcionado a la unidad de control durante al menos un mensaje, siendo posible leerlo hasta que el siguiente mensaje sea completado. Tiene que ser posible generar como mínimo un evento programable a partir de la base temporal. El evento debe ser libremente programado por la unidad de control. La planificación de las ranuras temporales de transmisión se divide en eventos. La retransmisión automática de mensajes que no pudieron ser transmitidos correctamente debe ser deshabilitada. El ciclo básico define la longitud de las ventanas transmitidas entre dos mensajes de referencia, los cuales representan el tiempo global, y se utiliza para la sincronización de los temporizadores locales. Existe una matriz o tabla que especifica la secuencia de mensajes transmitidos en cada ciclo básico [Führer00].

6.3 La arquitectura Time-Triggered

6.3.1 Introducción

Durante los últimos años, el uso de la arquitectura *Time-Triggered* (TTA) [Kopetz99, Kopetz03] se ha ido abriendo paso en la industria como una arquitectura para sistemas de tiempo real con una alta Confiabilidad. El uso industrial de esta arquitectura, principalmente en la industria aeronáutica y de automoción, hace necesaria la validación de la misma. Uno de los métodos más comunes para la validación de sistemas tolerantes a fallos es la inyección de fallos.

El protocolo *Time-Triggered* (*TTP*) es un protocolo de comunicaciones integrado, diseñado para arquitecturas *Time-Triggered*. Este protocolo proporciona los diferentes servicios requeridos para la implementación de sistemas en tiempo real tolerantes a fallos: transmisión de mensajes predecible, reconocimiento implícito de mensajes para comunicaciones en grupo, sincronización de reloj, servicio de pertenencia (del inglés *membership service*), cambios rápidos de modo y control de redundancia. La implementación de estos servicios se realiza sin mensajes extra y con una sobrecarga mínima en el tamaño del mensaje [Kopetz94].

A continuación, se va a presentar la arquitectura y el protocolo *Time-Triggered*, para después describir detalladamente los modelos en VHDL del controlador basado en esta arquitectura y utilizados durante los experimentos de inyección de fallos, los cuales se describirán en el capítulo siguiente.

6.3.2 El protocolo Time-Triggered

Como se ha comentado, el protocolo *Time-Triggered* (*TTP*) es un protocolo de comunicaciones de tiempo real utilizado en la interconexión de módulos electrónicos de sistemas distribuidos de tiempo real tolerantes a fallos. En esta tesis se ha trabajado con la versión *TTP/C* de este protocolo. Esta versión está adaptada para cumplir los requerimientos de la clase C para aplicaciones automovilísticas [SAE-C94].

El protocolo *TTP/C* controla el intercambio de mensajes entre diferentes módulos electrónicos conectados entre sí mediante una red común. Cada controlador *TTP/C* contiene sus propios datos de control, almacenados en una tabla personalizada denominada MEDL (en inglés *Message Descriptor List*). Esta tabla, predeterminada estáticamente antes de la puesta en marcha del sistema, especifica los instantes concretos de tiempo en los que el controlador debe enviar o recibir una trama.

Una red *TTP/C* está constituida por un conjunto de módulos electrónicos conectados entre sí mediante dos canales replicados, tal como se puede ver en la Figura 46 [TTP/C99]. Un *cluster*[33] está formado por una red *TTP/C* y los módulos electrónicos asociados.

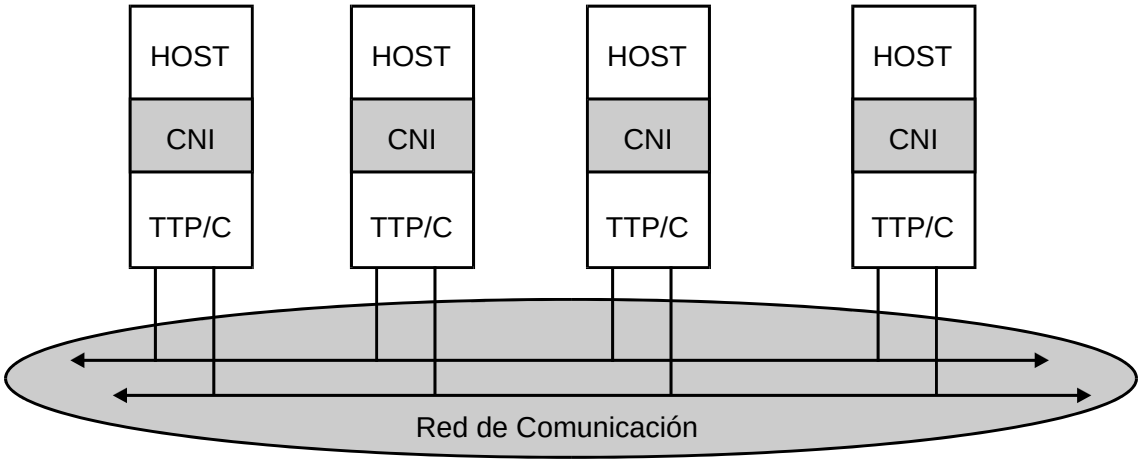


Figura 46. Un ejemplo de una red *TTP/C* [TTP/C99].

En una arquitectura *Time-Triggered*, se tiene un conocimiento previo sobre el comportamiento futuro del sistema. El protocolo *TTP/C* utiliza ampliamente esta información durante su funcionamiento. Por ejemplo, no es necesario enviar ninguna identificación dentro de un mensaje ya que el instante de tiempo en el que cada nodo del sistema debe transmitir un mensaje está predeterminado. De igual manera, un nodo puede determinar si ha ocurrido un fallo en la transmisión de un mensaje antes de que transcurra el tiempo necesario para la transmisión completa del mismo, sin intercambio de información alguna respecto al reconocimiento de mensajes [Kopetz94].

Una característica fundamental de la arquitectura *TTA* es el tratamiento del tiempo. Como se puede ver en la Figura 46, la arquitectura *TTA* está compuesta por un conjunto de nodos, proporcionando a cada nodo una base temporal global de precisión conocida tolerante a fallos. A partir de este conocimiento, la arquitectura *TTA* especifica de forma precisa las interfaces entre los nodos, simplifica la comunicación, establece la consistencia de estados[34], realiza una temprana detección de errores y soporta la temporización de aplicaciones de tiempo real [Maier02].

---

[33] Se define un *cluster* en [TTP/C99] como un conjunto de SRU que comparten un *bus* en un sistema *TTP/C*. Más adelante se explica qué es una SRU.

[34] Consistencia de estados: en un instante dado, los estados de todos los nodos son consistentes si dichos estados muestran el mismo valor para todos los nodos. Es decir, todos los nodos tienen la misma visión del sistema.

A partir de este conocimiento temporal, cada nodo de la arquitectura *TTA* genera sus propias señales de control a partir de su reloj local, evitando de esta manera el cruce de señales de control entre los distintos nodos.

La arquitectura *TTA* implementa la consistencia en las comunicaciones mediante *hardware* y a nivel de protocolo. Una de las ideas principales detrás del diseño del protocolo *TTP/C* es la consistencia en la transmisión de mensajes, siendo el propio protocolo el que garantiza dicha consistencia. Además, en caso de avería, el sistema de comunicaciones debe decidir independientemente cuál es el nodo erróneo, siempre que la hipótesis de fallos se mantenga. Estas propiedades se consiguen mediante tres mecanismos: pertenencia, reconocimiento y el algoritmo de *clique avoidance*. La combinación de estos tres mecanismos junto con la base temporal común, establecida mediante la sincronización del reloj, proporciona la consistencia en la comunicación. El protocolo *TTP/C* garantiza que todos los nodos que funcionen correctamente recibirán la misma información. De esta manera, la arquitectura *TTA* proporciona a la aplicación *software* un modelo de programación muy poderoso para manejar eficientemente sistemas complejos distribuidos. A continuación se hará una breve descripción de los tres mecanismos mencionados anteriormente [Maier02].

Mecanismo de pertenencia (en inglés *membership mechanism*). Cada nodo de un *cluster TTP/C* mantiene una lista de pertenencia con todos los nodos que se considera que funcionan correctamente. Cada vez que se recibe un mensaje, se actualiza esta información (cada receptor la actualiza localmente en función de que la transmisión haya sido correcta o incorrecta). De esta manera, se refleja localmente la vista que cada nodo tiene del resto de nodos del sistema. Con cada transmisión, cada nodo comprueba la lista de pertenencia del nodo emisor, ya que está explícitamente incluida en la trama o escondida en el cálculo del CRC del mensaje.

Debido al estricto esquema de transmisión, cada nodo comprueba las listas de pertenencia del resto de nodos durante una ronda de envío de mensajes (más adelante se explica el método de transmisión y la terminología aplicada). Cualquier nodo con una lista de pertenencia diferente es considerado incorrecto. Esta característica asegura una vista consistente de todos los nodos.

Mecanismo de reconocimiento (en inglés *acknowledgment mechanism*). Después de una transmisión de datos, el nodo que acaba de enviar el mensaje debe recibir el reconocimiento de recepción correcta por parte del resto de los nodos. En el protocolo *TTP/C* este reconocimiento se realiza mediante la comprobación de la lista de pertenencia. El método es el siguiente: en la siguiente transmisión de un mensaje (realizada por otro nodo), el primer nodo comprueba la lista de pertenencia recién recibida. Si en esta lista, el primer nodo está activado, asume que el mensaje ha sido correcto, puesto que el resto de nodos lo ha recibido correctamente. Si en la lista de pertenencia hubiera estado desactivado, el nodo informa a la aplicación que el último mensaje enviado ha sido incorrecto y deja de transmitir mensajes hasta volver a reintegrarse en el sistema.

Mecanismo de *clique avoidance*. El mecanismo del *clique avoidance* trata aquellos fallos que están fuera de las hipótesis de fallos. Este mecanismo detecta fallos en múltiples componentes e inconsistencias, así como soporta la estrategia NGU[35]. Antes de la transmisión de un mensaje, este algoritmo comprueba si el nodo pertenece al mismo grupo que la mayoría del *cluster*. Si el nodo está con la minoría, ha ocurrido un fallo que está fuera de las hipótesis de fallos, lo que lleva a una inconsistencia. El protocolo *TTP/C* avisa a la aplicación *software* de este escenario de fallo (fallos múltiples e inconsistencias). La aplicación puede decidir las operaciones a realizar a continuación.

[35] NGU, del inglés *Never-Give-Up*. El protocolo *TTP/C* implementa la estrategia NGU para escenarios de fallos múltiples, que aunque están fuera de las hipótesis de fallos, también son tratados, de tal manera que si un nodo detecta fallos que las hipótesis de fallos no cubren, se informa a la aplicación. Ésta puede cambiar el modo de funcionamiento del nodo para funcionar en un entorno de silencio ante fallos.

Antes de continuar es necesario presentar algunas definiciones para comprender el funcionamiento de la red *TTP/C*:

- CNI (del inglés *Communication Network Interface*): Interfaz utilizada en la comunicación entre el controlador de comunicaciones y el *host* del nodo.
- SRU (del inglés *Smallest Replaceable Unit*): Módulo electrónico conectado a un canal *TTP/C* cuya estructura se muestra en la Figura 47.
- FTU (del inglés *Fault-Tolerant Unit*): Grupo de SRU que garantizan un funcionamiento correcto incluso en el caso de la avería de una de ellas.

El bloque básico de la arquitectura *TTA* es el nodo (ver Figura 47). Esta unidad está compuesta por un procesador con memoria, que es el que ejecuta las aplicaciones *software*, un subsistema de entrada/salida y el controlador de comunicaciones *Time-Triggered* [Maier02].

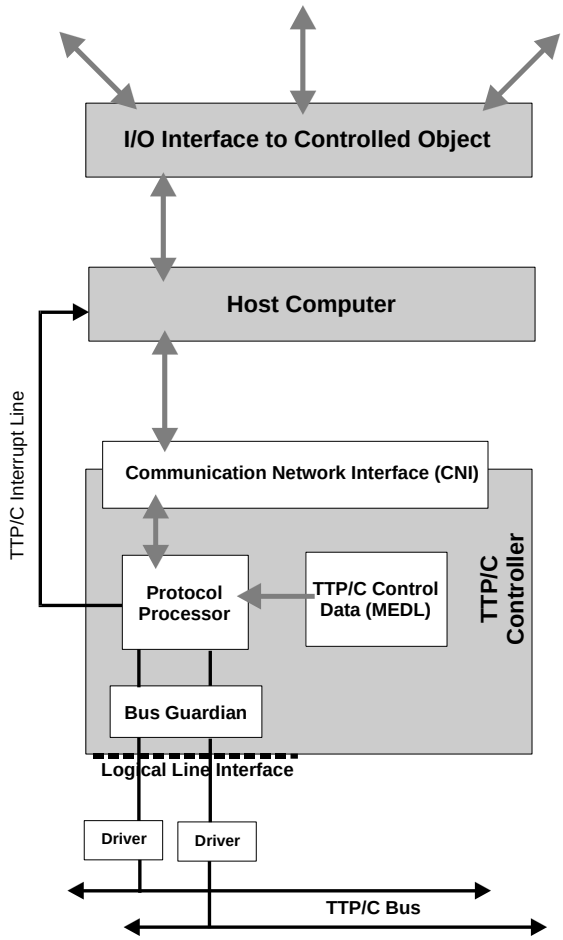


Figura 47. Diagrama de bloques de una SRU [TTP/C99].

La estructura del sistema *TTP/C* consiste en un conjunto de nodos *silenciosos ante fallos*[36] conectados entre sí mediante dos canales de comunicación replicados. Para tolerar fallos en los nodos, éstos se pueden replicar o se pueden agrupar en FTU. La arquitectura *TTA* garantiza que los nodos replicados realizan los mismos cambios de estado aproximadamente en el mismo instante de tiempo. Así mismo, una FTU se considera que está operativa si al menos uno de sus nodos está operativo. Los relojes locales de cada nodo se sincronizan con una precisión conocida, como se comentará más adelante. Cada nodo tiene un controlador de comunicaciones conectado con los dos canales replicados, así como mecanismos de detección de errores para la posible terminación de las operaciones en el caso de que se produzca un error [Kopetz94].

[36] Un nodo *silencioso ante fallos* (en inglés *fail-silent*) envía información correcta o no envía nada.

Dos canales de comunicación replicados conectan los nodos entre sí para formar un *cluster*. La estructura de interconexión física junto con los controladores de comunicaciones de todos los nodos conforman el subsistema de comunicaciones. Este subsistema lee el mensaje desde la CNI, incluyendo información del estado del nodo, en el instante predeterminado por la MEDL, la tabla de planificación que es conocida por todos los nodos del *cluster*. En dicho instante, conocido por todos los nodos del sistema, se iniciará la transmisión del mensaje [Maier02].

El acceso al *bus* está controlado por un esquema TDMA, basado en el reloj global que presenta el sistema. El esquema global de transmisión de los diferentes nodos se puede ver en la Figura 48. El funcionamiento es el siguiente. Cada SRU tiene un periodo predeterminado (en inglés *slot*) para enviar su trama, denominado *SRU slot*. Un conjunto de periodos de SRU (en inglés *SRU slots*) forma una ronda de TDMA (en inglés *TDMA round*), y a su vez, un conjunto de rondas de TDMA forma un ciclo de *cluster* (en inglés *cluster cycle*). La longitud de los diferentes ciclos (*TDMA round* y *cluster cycle*) está predeterminada de forma estática antes de poner en marcha el sistema, siendo conocida por todos los nodos del mismo.

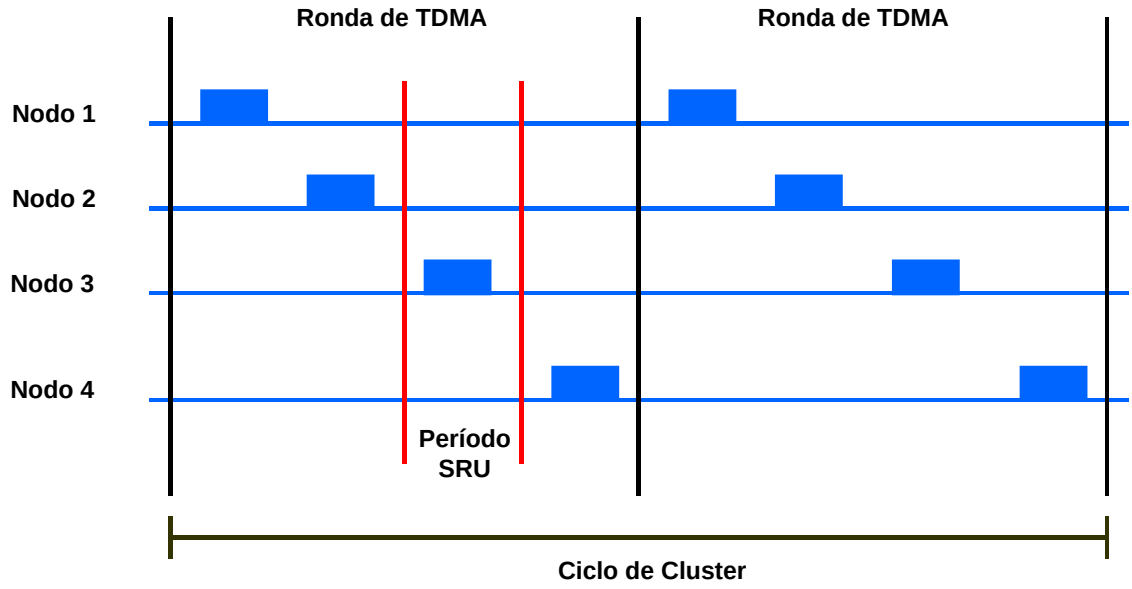


Figura 48. Esquema de transmisión en una red TTP/C [TTP/C99].

La Figura 49 muestra el funcionamiento particular del envío de mensajes en una red *TTP/C*. En este ejemplo, existen tres FTU, cada una de ellas formadas por dos SRU. Un periodo de FTU (en inglés FTU *slot*) está formado por dos periodos de SRU, en los que cada SRU envía el mensaje correspondiente. Un ciclo de *cluster* está formado por 3 periodos de FTU, con un total de 12 mensajes.

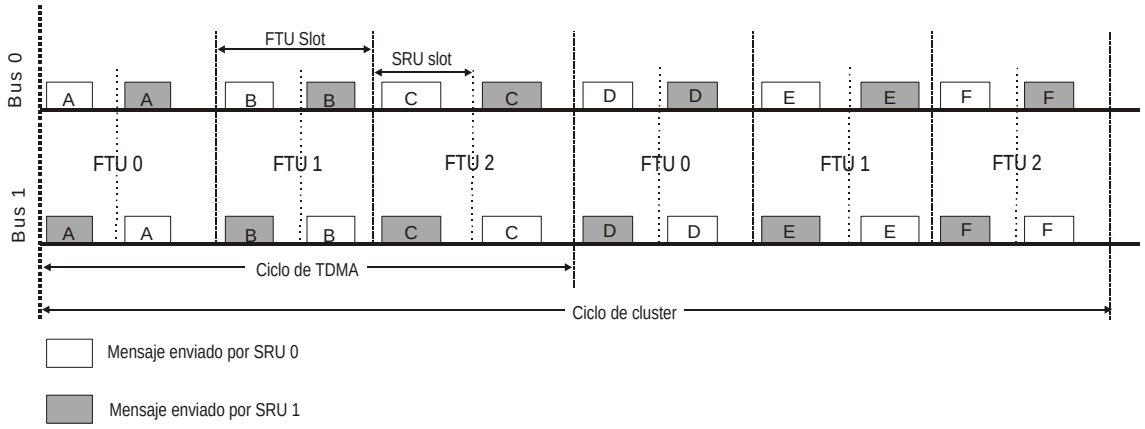


Figura 49. Funcionamiento del envío de mensajes en una red *TTP/C* [TTP/C99].

Físicamente, una SRU está compuesta por un *host*, un controlador y una interfaz de entrada/salida, tal y como puede verse en la Figura 47. La CNI (incluida en cada SRU) se utiliza como canal de comunicaciones entre el *host* y el controlador. Está formada por una RAM de doble puerto (DPRAM) y una línea entre el controlador y el *host* (línea de interrupción). La CNI contiene los datos que se intercambian el *host* y el controlador, actuando como interfaz entre éstos dos subsistemas.

La sincronización global del sistema es llevada desde el controlador al *host* mediante la línea existente entre ambos (línea de interrupción). Para reducir la carga del *host*, la interrupción sólo se activa en aquellos pulsos (llamados *ticks*) de la escala temporal global que son significativos para el *host*. Éste puede seleccionar los pulsos significativos mediante la programación de los parámetros adecuados en la CNI. Otra opción del *host* es la supresión de todos los pulsos.

La línea de interrupción también se usa para informar al *host* sobre diferentes condiciones excepcionales (por ejemplo, errores).

El controlador engloba al protocolo de comunicaciones, la MEDL y una unidad *hardware* independiente denominada Guardián del *Bus*, utilizada para proteger el *bus* de fallos de temporización del controlador.

Funcionamiento del protocolo *Time-Triggered*

El principio de operación del sistema es el siguiente. El controlador funciona autónomamente, sin ninguna señal de control desde el *host*. Toda la información de control necesaria está grabada en la MEDL, personalizada para cada controlador. El número de entradas de la MEDL se corresponde con la longitud del ciclo de cluster. La MEDL contiene los siguientes datos:

- Para cada mensaje, el instante en el que el mensaje debe ser transmitido y la dirección dentro de la CNI donde se tienen que buscar los datos.
- El instante en el que un mensaje debe ser recibido y la dirección del campo de datos en la CNI donde almacenar los datos recibidos.
- Información adicional para el funcionamiento del protocolo.

El protocolo *TTP/C* se aprovecha del conocimiento global de la planificación para realizar la sincronización del reloj. Cada nodo mide la diferencia de tiempo entre la llegada de un mensaje correcto esperado y su tiempo local. Con esta medición, cada nodo calcula la diferencia entre el reloj del emisor y el suyo propio. Un algoritmo tolerante a fallos utiliza esta información para calcular periódicamente un término de corrección que se aplica al reloj local con el fin de sincronizarlo con el resto de relojes del *cluster* [Maier02]. De esta manera, cuando el intervalo transcurrido alcanza un instante que está contenido en la MEDL, se ejecutan las acciones especificadas en la tabla.

Servicios del protocolo *Time-Triggered*

El protocolo *Time-Triggered* proporciona los siguientes servicios [Kopetz94, TTP/C99, Maier02]:

- Los retardos en la transmisión de todos los mensajes son predecibles y limitados, incluso en condiciones de máxima carga y en un escenario con fallos. El acceso al medio mediante una estrategia TDMA permite el envío autónomo de mensajes con un retardo conocido y un *jitter*[37] mínimo.
- Encapsulación temporal de subsistemas, de tal manera que es posible una metodología constructiva de prueba (o *test*) así como la exclusión de interferencia temporal de subsistemas desarrollados independientemente, gracias a las propiedades de la arquitectura de la comunicación.
- Rápida detección de fallos tanto en el receptor como en el emisor gracias al servicio de pertenencia o *membership service*. Este servicio proporciona una vista consistente de todos los nodos del sistema, tanto los activos como los inactivos. Cada nodo mantiene su visión del sistema mediante un registro (en inglés *membership vector*) en el que cada bit indica si un determinado nodo (o FTU) está activo o no. La actualización de este vector se realiza inmediatamente después del envío de un mensaje. Si el mensaje no ha llegado correctamente, los nodos receptores marcan al nodo emisor como inactivo en el registro de pertenencia. Este registro es enviado con cada mensaje, por lo que el nodo emisor del mensaje incorrecto sabe que no está funcionando correctamente nada más acabar el envío del mensaje del siguiente periodo (o *slot*). Si un nodo no está activado en el registro de pertenencia, no enviará ningún mensaje.
- El servicio de pertenencia informa a cada nodo de la consistencia en la transmisión de datos. Este servicio distribuido de reconocimiento informa rápidamente al nodo si ha ocurrido un error en el sistema de comunicaciones, es decir, si se ha perdido la consistencia de estado.
- Reconocimiento implícito de mensajes.
- La arquitectura tolerante a fallos se puede implementar fácilmente gracias a la existencia de canales y SRU replicados, existiendo, además, un servicio distribuido de manejo de la redundancia.
- Posibilidad de cambios rápidos del modo de funcionamiento.
- El protocolo proporciona un servicio de sincronización de reloj mediante un algoritmo de sincronización distribuido tolerante a fallos[38], sin coste adicional en el tamaño del mensaje, sin incrementar el número de mensajes y sin depender de un servidor de tiempos centralizado.
- No existe ninguna restricción en el uso de la red ya que no se requiere ningún tipo de arbitraje.
- Alta eficiencia de datos debida a la mínima sobrecarga del protocolo.
- El algoritmo de *clique avoidance* detecta fallos que no están contemplados en la hipótesis de fallos y que son intolerables en el nivel de protocolo (por ejemplo, fallos múltiples).

[37] *Jitter*: se puede definir como la diferencia de tiempo entre la ocurrencia planificada de una acción y cuando ocurre realmente.

[38] Se realiza en tres fases. En la primera, se lee el valor del reloj del resto de nodos, en la segunda se analizan los datos para detectar errores y ejecuta la función de convergencia para calcular el valor de corrección del reloj local, y en la tercera si su reloj local no se ha desviado más que el valor límite, se aplica el valor de corrección al reloj local. Si se ha desviado más que ese valor límite, el nodo se desactiva [Kopetz97].

Hipótesis y manejo de los fallos

En un sistema basado en el protocolo *TTP/C* (y correctamente configurado), los diversos componentes se encuentran encapsulados en diferentes regiones de contención de fallos, y en el que cualquier componente puede tener un fallo en un modo arbitrario de avería. Bajo esta premisa, la probabilidad de dos averías concurrentes e independientes en el mismo componente es lo suficientemente remota como para considerarla como un evento extraño. Sin embargo, sí que se necesitan mecanismos de detección de errores que actúen rápidamente para asegurar que dos fallos simples consecutivos no se conviertan en fallos concurrentes. Los diferentes mecanismos de detección de errores presentes en el controlador *TTP* se explican en el punto 7.4.1.2.1.

Para fallos *hardware*, el protocolo *TTP/C* aísla y tolera fallos simples en cualquier nodo. El Guardián del *Bus* asegura que un nodo con fallos no perturbará el funcionamiento del sistema ni provocará el envío de tramas con valores erróneos o fuera de su ventana de transmisión. Este guardián garantiza que un nodo sólo puede enviar un mensaje en una ronda de TDMA, eliminando así el problema del *babbling-idiot* [Maier02].

6.3.3 El modelo VHDL del controlador TTP/C: TTP/C-C1 y TTP/C-C2

En este apartado se van explicar los dos modelos en VHDL del controlador *TTP/C* utilizados durante los diferentes experimentos de inyección. Estos modelos fueron utilizados por *Austria Microsystems*[39] para realizar la síntesis de dos *ASIC*[40] que implementan el controlador. En concreto, a estos integrados se los denomina como AS8201 para el controlador *TTP/C-C1* y AS8202 para el controlador *TTP/C-C2*.

6.3.3.1 El modelo VHDL del controlador TTP/C-C1

La Figura 50 muestra el diagrama de bloques del controlador *TTP/C-C1*[41]. Como se puede ver, éste está centrado en la Unidad de Control del Protocolo o PCU (del inglés *Protocol Control Unit*), que es el corazón del sistema. El modelo se completa con varios bloques funcionales de bajo nivel que implementan funciones críticas y propiedades *hardware* relacionadas con el protocolo: transmisión de mensajes, sincronización del reloj, cálculo del CRC, etc. La PCU controla la interacción de los distintos bloques funcionales de bajo nivel, ejecutando además mecanismos del protocolo de alto nivel (servicio de pertenencia, control de la redundancia, etc.). Esta unidad está implementada como un procesador *pipe-line* con memoria de instrucciones interna, la cual es cargada desde una *flash* EPROM externa durante la inicialización o puesta en marcha del sistema. La PCU también realiza la función de maestro del *bus* de registros síncrono interno, el cual conecta los diferentes bloques de bajo nivel. Todos éstos bloques comparten un único espacio de direcciones. Las transferencias entre registros se realizan mediante instrucciones *MOVE* de la PCU. Por último, hay que mencionar que el controlador tiene una arquitectura de 16 bits.

Una RAM de doble puerto en la interfaz del *host* contiene la CNI. El Guardián del *Bus* está integrado en el chip VLSI, conteniendo un oscilador local. El patrón de accesos estáticos o MEDL reside en una *flash* EPROM externa, que puede ser accedida y programada por el controlador.

---

[39] http://www.austriamicrosystems.com/
[40] ASIC: *Application Specific Integrated Circuit*.
[41] *C1* hace referencia a la primera versión del controlador *TTP/C*.

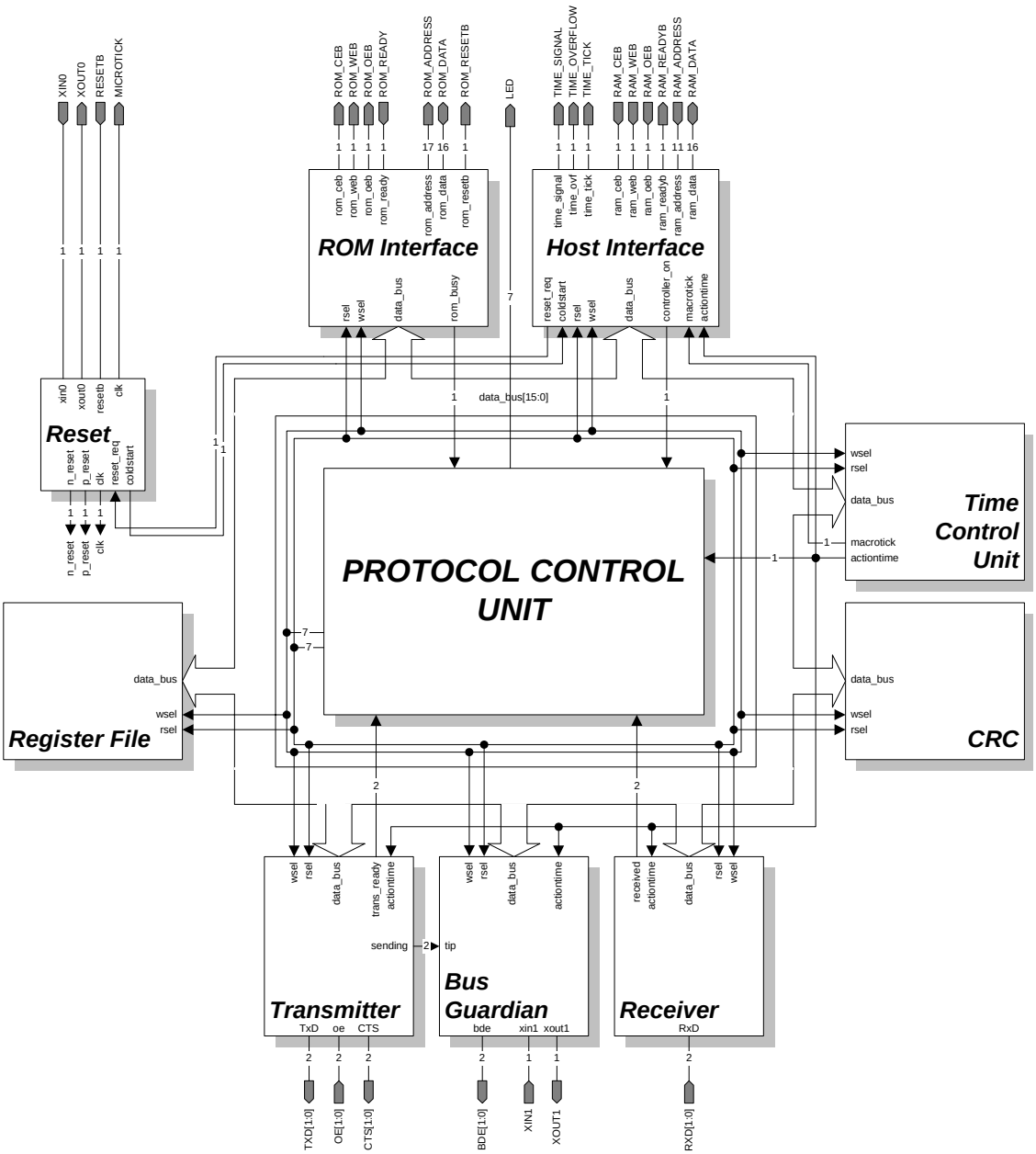


Figura 50. Diagrama de bloques del controlador TTP/C-C1 [Madritsch01]

A continuación, en los siguientes apartados se dará una explicación algo más detallada de los diferentes bloques del controlador.

6.3.3.1.1 Unidad de Control del Protocolo (Protocol Control Unit)

La PCU es un procesador con arquitectura *pipe-line* con una Unidad Aritmético-Lógica basada en el acumulador. Los programas se almacenan en una memoria de instrucciones que se carga desde la EPROM a través del *bus* de registros interno mediante un programa de arranque. Las instrucciones son de 16 bits y se ejecutan en la *pipe-line*, la cual está compuesta de tres etapas. La *pipe-line* resuelve situaciones de riesgo de datos y control autónomamente.

Formato de las instrucciones

La PCU puede ejecutar tres tipos de instrucciones:

- *ALU*: operaciones de la ALU entre registros o con valores inmediatos.

- *MOVE*: transferencias entre registros utilizando el *bus* de registros interno.
- *BRANCH*: control del flujo de programa.

La Figura 51 muestra el distinto formato de los diferentes tipos de instrucciones:

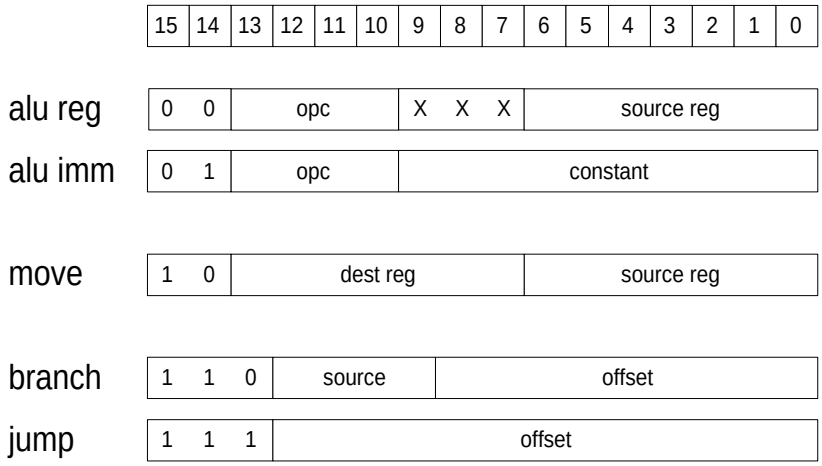


Figura 51. Formato de instrucciones [Sprachmann98].

Arquitectura de la unidad *pipe-line*

Como se ha comentado, la unidad *pipe-line* ejecuta las instrucciones en tres etapas:

1. IF (*Instruction Fetch*): El contador de programa direcciona la siguiente instrucción en la memoria de instrucciones. La memoria de instrucciones es accesible a través del *bus* de registros interno. En el caso de un acceso del *bus* de registros a la memoria de instrucciones o si se ha detectado un salto en el programa, la unidad *pipe-line* recibe una instrucción NOP. El contador de programa se puede leer y descargar a través del *bus* de registros interno.

   La ROM de arranque (del inglés *boot ROM)* contiene un programa que permite copiar el programa del protocolo de comunicaciones desde la *flash* EPROM hasta la memoria de instrucciones local.

2. DEC/MV/BAC (*Decode / Move / Branch Address Calculation*): Esta etapa decodifica la instrucción según el formato de la misma, distribuye las transferencias entre registros a través del *bus* de registros interno y alimenta la siguiente etapa de la unidad *pipe-line*.

   El nuevo valor del contador de programa depende del tipo de instrucción procesada en esta etapa. Para instrucciones de tipo *ALU*, el contador de programa se incrementa en uno. Existen dos tipos de instrucciones de tipo *BRANCH*: salto condicional e incondicional. En el segundo caso, se suma un desplazamiento al contador de programa. En el caso de saltos condicionales, esta etapa ejecutará el salto si la condición de salto incluida en la instrucción se cumple (mediante la suma de un desplazamiento al contador de programa).

   En las instrucciones de tipo *MOVE*, la trasferencia entre registros se ejecuta con la ayuda de dos señales internas del *bus* de registros interno.

   Para las instrucciones de tipo *MOVE* y *BRANCH* la etapa de decodificación suministra instrucciones *NOP* a la etapa EX. La unidad *pipe-line* permite el acceso en escritura y lectura al contador de programa a través del *bus* de registros interno, lo cual permite realizar llamadas a subprogramas.

3. EX (*Execute*): En esta etapa se ejecutan las instrucciones de tipo *ALU*. Dependiendo del tipo de operador, la Unidad Aritmético-Lógica obtiene los operandos de diferentes registros. El resultado de la operación se almacena en el acumulador.

Definición del juego de instrucciones

Se puede ver una definición formal del juego de instrucciones y el comportamiento de la unidad *pipe-line* para cada instrucción en la Tabla 24:

Etapa	Operación de transferencia de registros		
IF	PC ← Si *salto* entonces [PC + desplazamiento] si no [PC + 1]		
MV	ALU *register*	ALU *immediate*	MOVE
	rsel ← INSTR(6 downto 0); wsel ← RA_ALU; OPCODE ← INSTR(13 downto 10); CONST ← don't care; jump ← '0'; jump_offset ← don't care;	rsel ← RA_NONE; wsel ← RA_NONE; OPCODE ← INSTR(13 downto 10); CONST ← INSTR(9 downto 0); jump ← '0'; jump_offset ← don't care;	rsel ← INSTR(13 downto 7); wsel ← INSTR(6 downto 0); OPCODE ← NOP; CONST ← don't care jump ← '0'; jump_offset ← don't care;
BAC	branch	Jump	
	rsel ← RA_NONE; wsel ← RA_NONE; OPCODE ← NOP; CONST ← don't care jump ← branch(source, branch_vector); jump_offset ← sign_extent(INSTR(8 downto 0));	rsel ← RA_NONE; wsel ← RA_NONE; OPCODE ← NOP; CONST ← don't care jump ← '1'; jump_offset ← sign_extent(INSTR(12 downto 0));	
EX	ALU *register*	ALU *inmediate*	
	ACCU ← ACCU OPCODE OPERAND;	ACCU ← ACCU OPCODE exten(CONST,16);	

Tabla 24. Definición formal del juego de instrucciones [Sprachmann98].

El significado de cada celda de la tabla anterior es el siguiente:

- En la etapa IF se prepara la actualización del PC: si la instrucción es de salto incondicional, se añadirá el desplazamiento al PC. Si es otro tipo de instrucción, el PC se incrementará en uno (apuntará a la siguiente instrucción).
- En la etapa MV se ejecutan distintos algoritmos dependiendo del tipo de instrucción:
  - ⇨ Si la instrucción es de tipo *ALU register* (el dato está en un registro), actualiza *rsel*[42] con la dirección del registro fuente, *wsel*[43] con la dirección del acumulador y se extrae el código de operación (OPCODE).
  - ⇨ Si la instrucción es de tipo *ALU immediate* (el dato está en la instrucción), *rsel* y *wsel* no se utilizan. En este caso, se extraen el código de operación (OPCODE) y el dato inmediato (CONST) de la propia instrucción.
  - ⇨ Si la instrucción es de tipo *MOVE*, solamente se actualiza *rsel* (con la dirección del registro de lectura) y *wsel* (con la dirección del registro de escritura).
- En la etapa BAC se ejecutan los distintos tipos de saltos o cambios en el flujo del programa:
  - ⇨ Si la instrucción es de tipo *branch* (salto condicional), se ejecuta la función **`branch(source,branch_vector`**) para determinar si se ejecuta el salto o no, extrayendo de la propia instrucción el desplazamiento (**`jump_offset ← sign_extent(INSTR(8 downto 0));`**).
  - ⇨ Si la instrucción es de tipo *jump* (salto incondicional), simplemente se extrae el desplazamiento de la instrucción (**`jump_offset ← sign_extent(INSTR(12 downto 0));`**).
- En la etapa EX se ejecutan las instrucciones de tipo *ALU*:
  - ⇨ Si la instrucción es de tipo *ALU register*, la operación se realiza entre el acumulador y el registro indicado en la instrucción.

[42] Dirección del registro de lectura.
[43] Dirección del registro de escritura.

⇨ Si la instrucción es de tipo ALU *immediate*, la operación se realiza entre el acumulador y el dato almacenado en la instrucción.

Interfaz entre la memoria de instrucciones y el *bus* de registros

La memoria de instrucciones también está conectada al *bus* interno de registros, por lo que se pueden realizar funciones de carga y prueba. La interfaz consiste en el registro de direcciones de memoria, el registro de datos y la lógica de control necesaria para suministrar a la unidad *pipe-line* operaciones NOP en el caso de accesos a memoria. La lógica de control también es la responsable de mantener el valor del contador de programa cuando la memoria de instrucciones está siendo utilizada por el *bus* interno de registros. Tanto la lectura como la escritura en la memoria de instrucciones se ejecutan en un ciclo de reloj.

6.3.3.1.2 Banco de Registros (Register File)

Este módulo está formado por un grupo de registros de uso general. Comparte el mismo espacio de direcciones que los registros especiales de los diferentes bloques funcionales. Está compuesto por 64 registros de 16 bits. La latencia en las operaciones de escritura es de un ciclo de reloj.

6.3.3.1.3 Unidad de Interfaz con el Host (Host Interface Unit)

Este bloque está dividido en tres partes: la lógica de control de interrupciones y dos interfaces, una entre la DPRAM y el *bus* interno de registros (interfaz interno) y otra entre la DPRAM y el *host* (interfaz externa).

Lógica de control de interrupciones

Esta unidad se encarga de controlar las distintas interrupciones del sistema:

- Señal de reloj. Mediante la activación del registro de estado de las interrupciones, se generan eventos en la línea del reloj.
- Las interrupciones de los temporizadores se disparan cuando los registros de los temporizadores del *host* son iguales al registro global del tiempo.
- El bit de error del BIST[44] es activado cuando por lo menos uno de los bits del registro de estado del BIST es activado.

El registro de estado de las interrupciones acumula distintas interrupciones, sin borrar activaciones previas si éstas no han sido procesadas. Las distintas interrupciones pueden ser divididas en:

1. Eventos asíncronos:
   - Errores de BIST
   - Errores de Protocolo
   - Errores del *Host*
2. Eventos síncronos relacionados con el *macrotick*[45]:
   - Interrupción del temporizador 1
   - Interrupción del temporizador 2
3. Eventos síncronos relacionados con *actiontime*[46]
   - Pérdida de pertenencia

[44] *Built-In Self-test Mechanisms* o Mecanismos Internos Autocomprobantes.
[45] *Macrotick*: señal periódica que determina la granularidad del tiempo global.
[46] *Actiontime*: indica el comienzo planificado de una trama.

- Cambio en el vector de pertenencia
- Cambio de modo
- Interrupción 1 definida por el usuario
- Interrupción 2 definida por el usuario

Interfaz interna

Las operaciones de lectura y escritura en la DPRAM comienzan mediante una escritura en el registro de direcciones correspondiente. Un ciclo de lectura se inicia mediante la escritura en el registro de direcciones de lectura. Los datos estarán disponibles en el siguiente ciclo, y se podrán leer desde el registro de datos de la DPRAM. El ciclo de escritura se inicia al escribir en el registro de direcciones de escritura. Los datos para las operaciones de escritura deben ser escritos anteriormente en el registro de datos de la DPRAM.

Interfaz externa

En este caso, el aspecto de las operaciones de lectura y escritura es el de dos operaciones de este tipo sobre una memoria. Un ciclo de lectura comienza con la activación de *ceb*[47] mientras *web*[48] está desactivada. Como cualquier operación de lectura, los datos aparecerán en el *bus* interno un tiempo después de la estabilización de las direcciones. La operación de escritura es similar, pero teniendo en cuenta que hay que activar *web* y que los datos junto con la dirección tienen que estar estables antes de la escritura en sí.

6.3.3.1.4 Interfaz con la ROM (ROM Interface)

Este módulo maneja la comunicación entre el controlador *TTP/C* y la EPROM, así como el *reset* de la memoria. En este caso, está diseñado para cumplir las especificaciones de la memoria *flash* EPROM AM29F200 [ST99].

Operación de lectura

Se realiza en tres pasos:

1. Se escribe la dirección en el registro de direcciones.
2. Se espera hasta que el dato esté listo.
3. Se lee el dato desde el registro de datos.

Operación de escritura

Las operaciones de escritura se utilizan para tres tipos de acceso a la *flash* EPROM:

- Cambios de modo
- Escritura de datos
- Borrado parcial (sectores) o completo

Como en el caso de la lectura, la escritura de datos también se realiza en tres pasos:

1. Se escribe la dirección en el registro de direcciones.
2. Se escribe el dato en el registro de datos.
3. Se espera hasta que el dato esté escrito en memoria.

***Reset* externo**

Para ejecutar el *reset* de la memoria AM29F200, son necesarios los siguientes pasos:

1. Se escribe un **0** en el registro de *reset*.

[47] *ceb*: *chip enable*.
[48] *web*: *write enable*.

2. Se espera un mínimo de 500 ns.
3. Se escribe un **1** en el registro de *reset*.
4. Se espera durante 20 μs.

Contador de acceso de lectura/escritura

Este contador ajusta la velocidad del reloj del controlador al tiempo de acceso de la *flash* EPROM. El valor óptimo para el contador de acceso está dado por la siguiente fórmula:

$$contador = \frac{AT_{EPROM}}{CT_{controlador}} + 1$$

donde AT_{EPROM} es el tiempo de acceso a la EPROM y

$CT_{controlador}$ es el tiempo de ciclo del controlador

6.3.3.1.5 Unidad de Control Temporal (Time Control Unit)

La Unidad de Control Temporal o TCU (denominada *Time Control Unit* en la Figura 50) genera el *macrotick* sincronizado. Dependiendo del término de corrección, el *macrotick* es alargado o recortado en un *microtick*[49]. Esta corrección se realiza hasta que el término de corrección es cero. Después de un *reset*, la TCU está en modo pasivo y la generación del *macrotick* no está activa.

6.3.3.1.6 Unidad de CRC

Este módulo calcula el CRC de cada mensaje. Este cálculo se realiza utilizando dos polinomios diferentes (uno para cada canal), permitiéndose además el cálculo concurrente de ambos polinomios.

El cálculo del CRC comienza mediante una escritura en el registro de datos. Es posible iniciar cálculos consecutivos del CRC cada ciclo de reloj. La latencia en el cálculo del CRC es de dos ciclos de reloj.

6.3.3.1.7 Receptor (Receiver)

Esta unidad se encarga de la recepción de mensajes a través del *bus* del sistema. El receptor no funcionará si no es correctamente inicializado después de un *reset*. Una vez que se ha inicializado correctamente, el receptor analiza permanentemente las líneas del *bus* para comprobar si hay tráfico. Una vez que se ha recibido la cantidad esperada de *bytes*, el receptor asume que el mensaje está completo y espera el siguiente mensaje.

Cuando se recibe un dato, esta unidad activa una señal, siempre que este dato cumpla una de las siguientes condiciones:

1. Se ha recibido la cabecera o el primer byte de datos de un mensaje.
2. El tiempo marcado para la recepción de un mensaje ha pasado sin haberse recibido nada (o sin señales válidas en el *bus*).

Por otra parte, esta señal es desactivada cuando la PCU lee el dato recibido.

Durante la recepción de un mensaje, este módulo realiza varias comprobaciones en la señal de transmisión. Como se ha comentado en el punto 6.3.2, un mensaje se enviará o se recibirá en ciertos instantes predefinidos de tiempo (marcados por la MEDL). En el caso de la recepción, en realidad lo que existe es una ventana para realizar esta recepción. La Figura 52 muestra la llegada de tramas válidas e inválidas. Hay que mencionar la existencia de un delimitador entre

[49] *Microtick*: señal periódica generada por el oscilador del controlador.

la llegada de dos tramas, denominado IFG o *InterFrame Gap*, utilizado por el receptor y el transmisor para realizar operaciones internas.

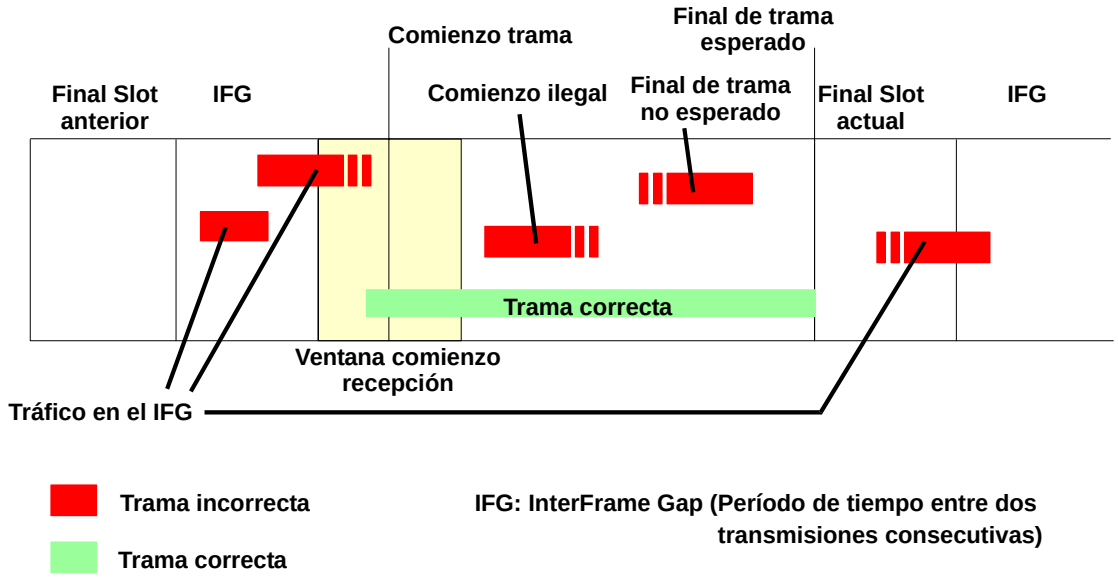


Figura 52. Llegada de mensajes válidos/inválidos [Sprachmann98].

6.3.3.1.8 Transmisor (Transmitter)

Este módulo se encarga de transmitir el mensaje de su nodo a través del *bus* mediante la codificación MFM (*Modified Frequency Modulation*). Si el mensaje no está preparado, no se envía nada, siendo el resto de nodos del sistema los que detectarán el silencio en el *bus*.

6.3.3.1.9 Guardián del Bus (Bus Guardian)

Este módulo es un dispositivo autónomo que protege los canales de un fallo en la temporización en un controlador. Contiene un oscilador propio para poder sortear o evitar posibles averías en el reloj del controlador. Se puede ver al Guardián del *Bus* como un refuerzo para la protección del *bus*. El Guardián del *Bus* realiza el control para que las operaciones sean correctas en el dominio temporal, por lo que éste módulo debe mantener un conjunto de parámetros con los que realizar dicho control.

6.3.3.1.10 Bus de registros interno

Este componente conecta los diferentes bloques funcionales del modelo, tal como se puede ver en la Figura 53. La PCU (*Protocol Control Unit*) gobierna las transferencias de datos entre los distintos bloques funcionales, actuando como el maestro del *bus*. Además, la PCU proporciona dos direcciones a cada bloque, una para operaciones de lectura y otra para operaciones de escritura (la lectura y la escritura se realizan desde el punto de vista de la PCU).

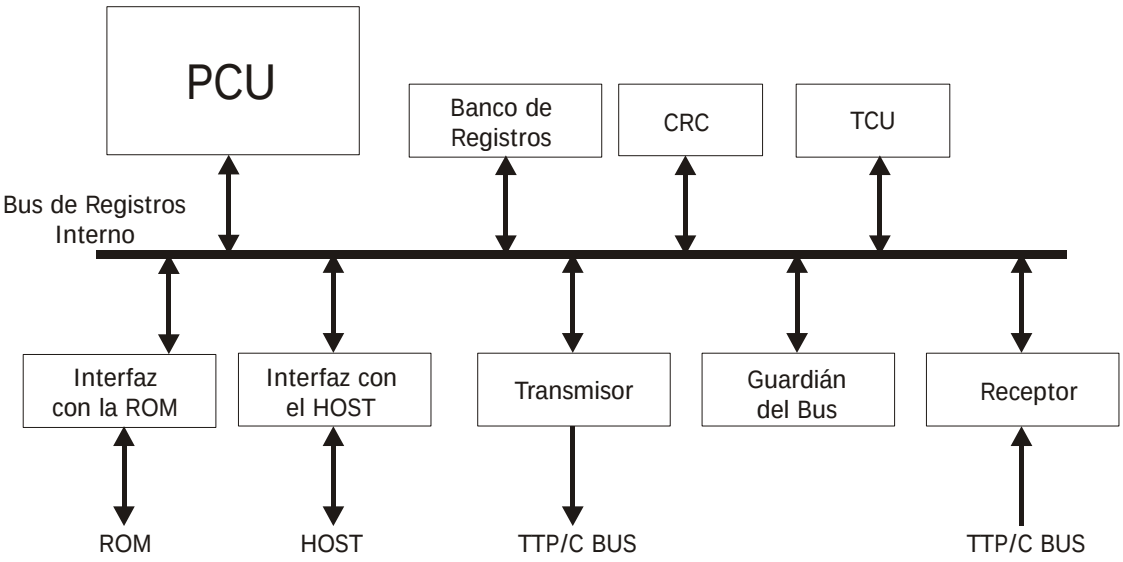


Figura 53. Arquitectura del controlador TTP/C-C1 [Gracia03b].

6.3.3.2 El modelo VHDL del controlador TTP/C-C2

La Figura 54 muestra el diagrama de bloques del controlador *TTP/C-C2*. Como se puede observar, es muy parecido al *TTP/C-C1* (cuyo diagrama de bloques se muestra en la Figura 50). Como en el caso del *TTP/C-C1*, la funcionalidad del protocolo está centrada en la Unidad Central del Protocolo o PCU. También existen diferentes bloques funcionales que realizan funciones críticas o relacionadas con el *hardware*, como el cálculo del CRC, transmisión de tramas, sincronización del reloj, etc.

Las diferencias entre ambos controladores se pueden dividir en dos: físicas y funcionales. Las principales las diferencias físicas se pueden resumir en cuatro puntos:

- El controlador *TTP/C-C1* permite una tasa de transferencia de hasta 2 Mbits/s con un reloj a 20 MHz, mientras que el *TTP/C-C2* permite una tasa de transferencia de hasta 5 Mbits/s de forma asíncrona y de hasta 25 Mbits/s de forma síncrona con un reloj a 40 MHz.
- Aunque ambos controladores están fabricados con tecnología CMOS, el *TTP/C-C1* es de 0.6 μm mientras que el *TTP/C-C2* es de 0.35 μm.
- Para el controlador *TTP/C-C1* es necesaria una fuente de alimentación de 5 V, mientras que para el *TTP/C-C2* es de 3.3 V.
- La última gran diferencia física es la memoria interna del integrado. Si en el *TTP/C-C1* existe una RAM de 1K x 16 que se utiliza para almacenar datos de control, de estado y mensajes, en el *TTP/C-C2* esta RAM interna es de 2K x 16, y además, también están integradas una SRAM de 4K x 16 para instrucciones y datos de configuración, una ROM de arranque de 4K x 16 y una FLASH interna de 16K x 16 para el microcódigo y para información de planificación.

En cuanto a las diferencias funcionales, estas se presentan en 3 nuevos módulos. Estos son la Unidad de Interfaz con el *Bus* (del inglés *Bus Interface Fifo*), el Banco de Registros (del inglés *Register File*) y la Interfaz con la Memoria FLASH Empotrada (del inglés *Embedded FLASH Interface*). A continuación se hará un resumen de la funcionalidad de estos módulos, y no se comentará nada de aquellos módulos que presentan una funcionalidad idéntica a los correspondientes del controlador *TTP/C-C1*.

La Unidad de Interfaz con el *Bus* está compartida por los bloques encargados de la recepción y la transmisión. Estos bloques no están activos nunca al mismo tiempo para un mismo canal, es decir, no hay una multiplexación de funciones en el tiempo para un canal dado. La Unidad de Interfaz con el *Bus* es utilizada como un almacén temporal para tramas de entrada o salida, permitiendo diferentes tipos de operaciones en el sistema: un canal recibiendo y el otro transmitiendo, ambos canales transmitiendo/recibiendo, canales síncronos, etc.

A diferencia del *TTP/C-C1*, el nuevo interfaz con el *bus* soporta transmisiones de 25 Mbits/s (el *TTP/C-C1* puede transmitir a 2 Mbits/s con una frecuencia de reloj de 20 MHz). El almacén temporal está implementado mediante una FIFO que, al igual que el transmisor y receptor, está diseñada de tal forma que permite utilizar ambos canales con distintos modos de operación. Este hecho permite a la PCU escribir y leer datos mientras la FIFO de la Unidad de Interfaz con el *Bus* está transmitiendo y recibiendo.

El banco de registros generales comparte el mismo espacio de direcciones que los registros especiales de los diferentes bloques funcionales. Está compuesto por 96 registros de 16 bits, los cuales están divididos en 3 páginas de 32 registros cada página, a diferencia del *TTP/C-C1*, cuyo banco de registros está compuesto por 64 registros de 16 bits.

La Interfaz con la Memoria FLASH Empotrada proporciona el acceso a la memoria FLASH empotrada mediante operaciones de transferencia de registros. Suministra también todas las señales de control y la temporización necesaria para operar e incluso comprobar la memoria FLASH empotrada.

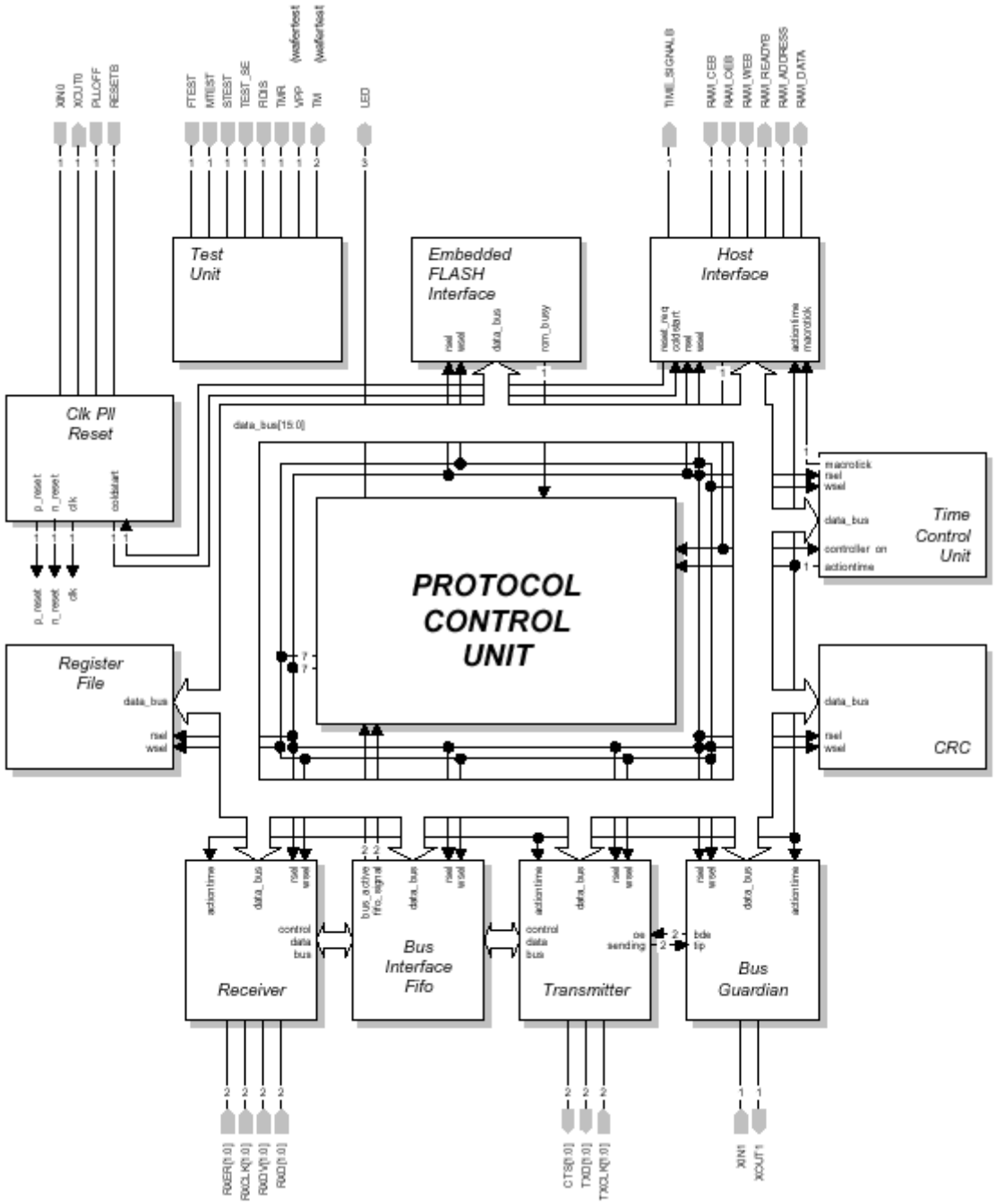


Figura 54. Diagrama de bloques del controlador TTP/C-C2 [Ley01].

Por último, la Figura 55 muestra el *Bus* de Registros Interno, el cual conecta los distintos bloques funcionales del modelo. La transferencia de datos entre los bloques funcionales está controlada por la PCU, actuando como el maestro del *bus*, tal y como ocurre en el *TTP/C-C1*.

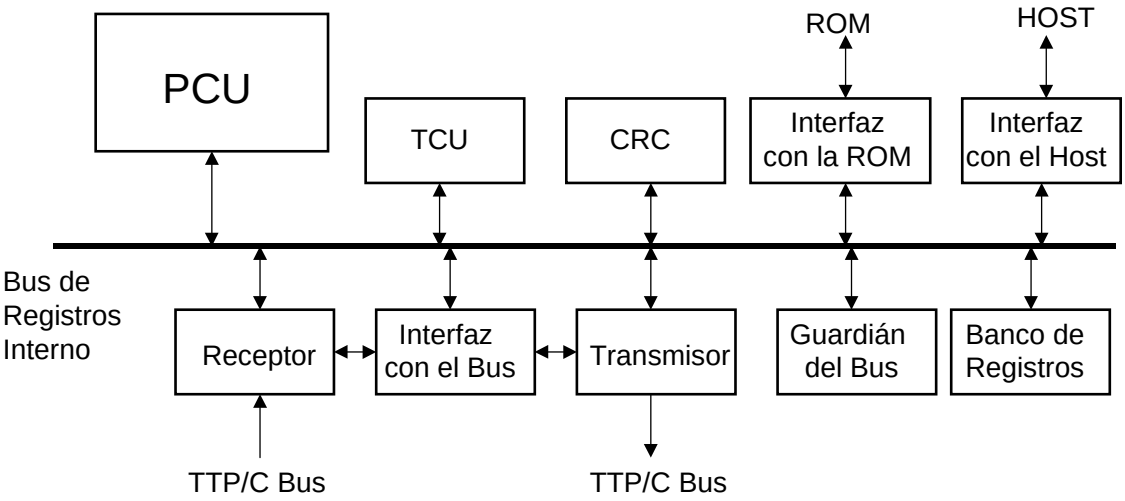


Figura 55. Arquitectura del *bus* de registros interno.

6.4 Resumen

En este capítulo se han presentado las características generales de las arquitecturas basadas en *buses*, incidiendo en los *buses* basados en el tiempo o *Time-Triggered*. Estos sistemas son los que actualmente se están introduciendo en la industria de aeronáutica y de automoción. A continuación se han descrito brevemente varias de estas arquitecturas, para después describir con detalle la arquitectura *Time-Triggered*.

El protocolo *Time-Triggered* es un protocolo de comunicaciones integrado, diseñado para arquitecturas *Time-Triggered*. Este protocolo proporciona los diferentes servicios requeridos para la implementación de sistemas distribuidos de tiempo real tolerantes a fallos: transmisión de mensajes predecible, reconocimiento implícito de mensajes para comunicaciones en grupo, sincronización de reloj, servicio de pertenencia, cambios rápidos de modo y control de redundancia. La implementación de estos servicios se realiza sin mensajes extra y con una sobrecarga mínima en el tamaño del mensaje.

Para finalizar, se han presentado los modelos en VHDL de los controladores *TTP/C-C1* y *TTP/C-C2*. El interés de estos modelos radica en que han sido utilizados durante la síntesis en *ASIC*, y en su utilización en diversos experimentos de inyección de fallos, los cuales han servido para validar la arquitectura *Time-Triggered*. Estos experimentos de inyección se detallan en el capítulo siguiente.

7 Validación de sistemas distribuidos de tiempo real tolerantes a fallos para aplicaciones críticas

7.1 Introducción

En el capítulo anterior se ha presentado la arquitectura *Time-Triggered* así como dos versiones en VHDL del controlador de comunicaciones *TTP/C*. Este capítulo se centra en la validación de esta arquitectura mediante la inyección de fallos en VHDL. Antes de describir los diferentes experimentos de inyección de fallos, se hará un breve estudio de la validación de sistemas distribuidos de tiempo real tolerantes a fallos, basándonos en los diferentes proyectos patrocinados por la Unión Europea, para después describir el proyecto *FIT* (*Fault Injection in the Time-Triggered Architecture* [Madritsch01]). Es en el marco de este proyecto en el que se realizaron los experimentos de inyección de fallos.

7.2 Validación de sistemas distribuidos de tiempo real tolerantes a fallos

Uno de los principales problemas que presenta la introducción de sistemas distribuidos de tiempo real tolerantes a fallos para aplicaciones críticas es su validación. Para realizar esta validación existen varios métodos. En primer lugar, se pueden utilizar métodos formales, que implican el uso de técnicas matemáticas y/o lógicas para describir, investigar y analizar la especificación, diseño, documentación y comportamiento del *hardware* y/o del *software*. El segundo método es la prueba (del inglés *test*), en el que al sistema *hardware* y/o *software* se le introducen valores con el fin de determinar si el sistema proporciona todas las funciones especificadas. Por último, se puede utilizar la inyección de fallos [Kopetz97].

Como prueba de la importancia de la validación, varios proyectos europeos han incidido en esta problemática. En el proyecto denominado *"X-By-Wire. Safey Related Fault Tolerant Systems in Vehicles"* [Dilger97, XbyWire98] se introdujo la arquitectura necesaria para la implementación de los sistemas *x-by-wire*. El objetivo de este proyecto era la definición de un marco de trabajo para la introducción de sistemas electrónicos tolerantes a fallos para aplicaciones críticas (sobretodo en la industria automovilística), teniendo en cuenta que estos nuevos sistemas no incluyen mecanismos mecánicos o hidráulicos.

En este proyecto se puso de manifiesto que la mejor opción es una arquitectura basada en el tiempo, y se optó por la arquitectura *Time-Triggered* [Kopetz98a, Kopetz98b]. Una vez que la arquitectura de los sistemas *x-by-wire* está clara, el siguiente paso es su validación. Un primer intento se presenta en este mismo proyecto [Hedenetz98, XbyWire98], donde realizan la inyección de fallos mediante la perturbación de los canales de transmisión, utilizando para tal fin un inyector físico. El propósito es la evaluación del comportamiento de los componentes tolerantes a fallos de la arquitectura *Time-Triggered*.

Sin embargo, son tres proyectos diferentes los que realmente inciden en esta problemática: *PDCS*, *PDCS-2* y *FIT*. El proyecto *PDCS* (del inglés *Predictably Dependable Computing Systems*) tenía como objetivo el producir un entorno de ayuda para el diseño de sistemas distribuidos de tiempo real tolerantes a fallos. Para ello dividieron el trabajo en tres grandes tareas:

- Definición de conceptos y terminología de la Confiabilidad.
- Especificación y diseño para una alta Confiabilidad.
- Evaluación y predicción de la Confiabilidad.

En este proyecto se identificaron los problemas para la validación de sistemas de tiempo real tolerantes a fallos y se propusieron una serie de atributos a la hora de utilizar la inyección de fallos [Arlat92b].

Continuando el trabajo de este proyecto, el *PDCS-2* profundizó en los puntos del proyecto *PDCS* e investigó en la mejora del proceso de diseño e implementación de sistemas distribuidos de tiempo real tolerantes a fallos, y especialmente cuando se necesitan grandes requerimientos de Confiabilidad. El programa de investigación planeado incluía estudios en el campo de prevención, eliminación y predicción de fallos, y tolerancia a fallos [Randell95]. En resumen, los objetivos de los proyectos *PDCS* y *PDCS-2* fueron hacer que el proceso de diseño e implementación de sistemas confiables fuera mucho más predecible y con un coste menor de lo que se estaba haciendo hasta ese momento.

Para llevar a cabo los diferentes experimentos de inyección de fallos [Folkesson99, Arlat03b], se utilizó la arquitectura *MARS* (del inglés, *Maintainable Real-Time System*) [Kopetz82, Reisinger94, Karlsson95, Reisinger95], antecedente de la arquitectura *Time-Triggered* [Kopetz98a, Kopetz98b, Yuste99].

El objetivo de los experimentos de inyección de fallos en la arquitectura *MARS* fue el determinar experimentalmente la cobertura de detección de errores de ésta. En dicha arquitectura se implementaron tres niveles de mecanismos de detección de errores (EDM), los cuales proporcionaban la propiedad del silencio ante fallos en los nodos *MARS*. Estos mecanismos eran [Kopetz97]: (i) EDM *hardware*, (ii) EDM *software*, concretamente implementados en el sistema operativo y en el *software* de soporte, y (iii) EDM implementados en el nivel más alto, en este caso el nivel de aplicación.

Se utilizaron cuatro técnicas de inyección de fallos, divididas en dos grupos de experimentos. El primer grupo de experimentos [Karlsson95], se realizó mediante tres técnicas de inyección de fallos *hardware*: inyección de fallos física a nivel de *pin*, inyección de fallos basada en iones pesados e inyección de fallos mediante interferencias electromagnéticas (ver los puntos 2.2.2.1.1, 2.2.2.2.1 y 2.2.2.1.2).

Los resultados mostraron que la mayoría de errores eran detectados por los EDM *hardware*, seguidos por los EDM *software*. Aunque los EDM en el nivel de aplicación fueron los que detectaron una menor cantidad de errores, sin embargo, cuando no fueron habilitados, la cobertura del silencio ante fallos se redujo, especialmente con la técnica de los iones pesados, lo que mostró la necesidad de utilizar también estos mecanismos.

Los resultados obtenidos también sirvieron para identificar semejanzas y diferencias en los conjuntos de errores generados por las tres técnicas. Los conjuntos de errores se observaron indirectamente mediante la distribución de los errores detectados por los diferentes EDM. Se observó que la inyección de fallos a nivel de *pin* activó principalmente los EDM *hardware*, mientras que los iones pesados y las EMI ejercitaron principalmente los EDM *software* y los EDM implementados en el nivel de aplicación. Estos resultados llevaron a la conclusión que las diferentes técnicas son complementarias entre sí.

El segundo grupo de experimentos se realizó mediante la inyección de fallos implementada por *software* (o SWIFI, ver punto 2.2.3) [Fuchs96]. Después de realizar una serie de experimentos con esta técnica, se dedujeron las siguientes conclusiones:

- Se obtuvieron coberturas de detección similares utilizando SWIFI en el segmento de código y utilizando las técnicas *hardware*.
- Los EDM del nivel de aplicación detectan más errores con SWIFI que con las técnicas *hardware*. Este es debido principalmente a que los errores inyectados con las técnicas *hardware* no se propagan al nivel de aplicación, ya que son detectados principalmente por los EDM *hardware* y *software*.
- Si se desactivan los EDM del nivel de aplicación, SWIFI genera un número mayor de violaciones que EMI o la inyección a nivel de *pin*.

- En resumen, el conjunto de errores generados por SWIFI es capaz de producir un conjunto de errores similares a los producidos por EMI y la inyección a nivel de *pin*. Sin embargo, la inyección mediante iones pesados requiere unos modelos superiores (o más maliciosos) que el modelo del *bit-flip*, que es el utilizado por SWIFI.

Sin embargo, el esfuerzo más grande en la validación de este tipo de sistemas se ha realizado en el proyecto *FIT* (del inglés *Fault Injection for TTA*). El objetivo de este proyecto fue la validación experimental de los conceptos de la arquitectura *Time-Triggered* [FIT02a, FIT02b, FIT02c]. En el punto 7.3 se puede ver información más detallada de este proyecto, mientras que en el punto 7.4 se describen una serie de experimentos de inyección de fallos, que se realizaron dentro de este proyecto.

No obstante, antes del proyecto *FIT*, varios trabajos se han llevado a cabo para validar diferentes mecanismos y algoritmos de la arquitectura *Time-Triggered*. De estos trabajos podemos destacar varios. En primer lugar, en [Miner93, Schwier98] se verifica formalmente el protocolo de sincronización del reloj. La verificación formal del protocolo *Time-Triggered* se ha realizado en [Pfeifer99], mientras que el algoritmo de pertenencia[50] y el algoritmo del *clique avoidance* se han verificado formalmente en [Pfeifer00] bajo las hipótesis de fallos estándar de la arquitectura *Time-Triggered*. En [Rushby01c] también se ha verificado formalmente las reglas de temporización de los controladores y los Guardianes de *buses*. Por último, en [Rushby99, Rushby02a] se presentan varias verificaciones formales de diferentes propiedades de la arquitectura *Time-Triggered*.

7.3 El proyecto FIT: Fault Injection in the Time-Triggered Architecture

7.3.1 Objetivos del proyecto FIT

Como se ha comentado en la introducción, el principal objetivo del proyecto *FIT* fue la validación experimental de un sistema de comunicaciones basado en la arquitectura *Time-Triggered*. En particular, en la propuesta del proyecto se especifican los principales objetivos del mismo [FIT-A99, FIT-B99]:

1. Determinar la cobertura de detección de errores del sistema mediante el uso de diferentes técnicas de inyección de fallos, tanto *hardware* como *software* y utilizando una aplicación real.
2. Localizar debilidades en la arquitectura y buscar y evaluar alternativas de diseño con el fin de corregir estas debilidades.
3. Encontrar el conjunto de parámetros óptimos para los mecanismos de detección de errores bajo restricciones técnicas y económicas.
4. Comparar la efectividad de las diferentes técnicas de inyección de fallos para la evaluación de la tolerancia a fallos.
5. Desarrollar un entorno de pruebas para poder validar los diferentes mecanismos de fallos en distintas implementaciones en silicio del sistema.

Para realizar la validación experimental, se han utilizado diferentes técnicas de inyección de fallos [FIT02d]:

- Mediante el uso de un prototipo *hardware*:
  - ⇨ Inyección de fallos a nivel de *pin*. Inyecta los fallos mediante la perturbación de los *pines* del circuito integrado que implementa el controlador *TTP/C*.

[50] En inglés *membership algorithm*.

- ⇨ Inyección de fallos mediante el uso de iones pesados. Inyecta los fallos mediante el bombardeo con iones pesados del circuito integrado que implementa el controlador *TTP/C*.
- ⇨ Inyección de fallos implementada por *software*. Emula fallos *hardware* (en concreto *bit-flips* transitorios) mediante el *software*.
- ⇨ Inyección de fallos basada en el microcódigo del protocolo. Se modifica sistemáticamente el código y los datos del microcódigo del protocolo mediante la emulación de *bit-flips* permanentes[51].

- Mediante el uso de simulación:
  - ⇨ Inyección de fallos basada en VHDL. Se inyectan fallos mediante la modificación de los valores de las señales y variables del modelo en VHDL del controlador *TTP/C*.
  - ⇨ Inyección de fallos basada en *C-SIM*. Este método simula un modelo a nivel del protocolo del *TTP/C*. La inyección de fallos se realiza en cualquier área de memoria.

A continuación, se resumirán las distintas técnicas de inyección de fallos utilizadas durante el proyecto *FIT* [FIT-C99, FIT00a, FIT00b, FIT02d], sin entrar en detalles en los resultados conseguidos por las mismas, ya que éstos quedan fuera del ámbito de la presente tesis.

7.3.2 Técnicas de inyección de fallos del proyecto FIT

7.3.2.1 Inyección de fallos física a nivel de pin

Esta técnica se basa en la perturbación de los valores lógicos de los *pines* de los circuitos integrados, con el fin de provocar tanto fallos internos como externos. Puede ser utilizada para modificar datos de entrada y/o de salida. La mayor diferencia que presenta esta técnica respecto a otras técnicas de inyección de fallos *hardware* o implementadas por *software* es el gran control en la perturbación de las señales utilizadas para comunicar un circuito integrado con otros elementos del sistema [Martínez99, Blanc02a, PGil03]. Las perturbaciones típicas que se inyectan son indeterminaciones, puentes (del inglés *bridging*), pulsos (del inglés pulse) y *stuck-at*.

Durante el proyecto *FIT*, con esta técnica se inyectaron fallos en los *pines* del controlador de comunicaciones, los cuales se agruparon en:

- Interfaz con el controlador del *host*.
- Interfaz con la ROM.
- Señales del reloj.

7.3.2.2 Inyección de fallos mediante iones pesados

Esta técnica inyecta fallos transitorios del tipo *bit-flip* en localizaciones internas de los circuitos integrados, gracias a la radiación producida por una fuente de Californio-252. Esta radiación provoca que los fallos inyectados se distribuyan por todo el circuito. Para poder realizar la inyección de fallos, se debe quitar el encapsulado del circuito integrado y además, el circuito junto con la fuente radiactiva deben ser aislados en una cámara de vacío [Karlsson95, Sivencrona03].

[51] Aunque el *bit-flip* es un modelo de fallo intrínsecamente transitorio, en este caso se perturba una memoria que no va a ser escrita con posterioridad. Por tanto, el *bit-flip* inyectado permanecerá inalterado y de ahí viene el apelativo de permanente.

7.3.2.3 Inyección de fallos implementada por software

El objetivo de la inyección de fallos implementada por *software* es la emulación de fallos físicos y la inyección de fallos *software*. En particular, durante el proyecto *FIT* se inyectaron fallos en la CNI y el controlador del *host*, realizándose estas inyecciones de fallos en tiempo de ejecución (esta técnica se clasificaría como *run-time SWIFI*). Los fallos inyectados son emulaciones de *bit-flips* transitorios, siendo la localización de las inyecciones la arquitectura del sistema, no una aplicación en particular [Carreira98, Ademaj02]. En concreto, se han inyectado fallos en dos subsistemas:

- Dentro del controlador de comunicaciones *TTP/C-C1*, en la CNI, registros y memoria de instrucciones.
- Dentro del controlador del *host*, en la CNI y en la RAM del propio controlador.

7.3.2.4 Inyección de fallos basada en el microcódigo del protocolo

Esta técnica se clasificaría en el grupo de técnicas conocidas como *pre-runtime SWIFI*, e inyecta los fallos mediante la emulación de *bit-flips* permanentes simples o múltiples en el microcódigo del controlador. Este tipo de inyección de fallos representa fallos *hardware* permanentes dentro de las áreas de memoria (ROM o RAM), registros y *buses*. Esta técnica puede inyectar fallos de tres maneras [FIT01b]:

1. Aproximación lineal o secuencial (del inglés, *linear-scan approach*). En este caso, se modifican todos los bits del microcódigo en orden secuencial.
2. Aproximación aleatoria (del inglés, *random approach*), en la que se seleccionan un número de localizaciones estadísticamente significativas donde inyectar los fallos.
3. Aproximación sistemática (del inglés, *systematic approach*). Esta opción es la más favorable a la hora de seleccionar las localizaciones de la inyección. Se divide en tres pasos: i) Estudio del microcódigo para distinguir las instrucciones que se ejecutan realmente de las instrucciones que no se ejecutan. ii) Definición de las reglas utilizadas para alterar determinados bits. iii) Reprogramación del controlador con los fallos inyectados para estudiar el comportamiento del mismo.

7.3.2.5 Inyección de fallos basada en VHDL

Utilizando la técnica de los comandos del simulador, tal y como se ha desarrollado en los capítulos 1 y 1, se inyectaron fallos transitorios de diversos tipos. No se inyectaron fallos permanentes debido a que las especificaciones del proyecto *FIT* marcaban sólo la inyección de fallos transitorios. Gracias al VHDL y a la posibilidad de modelar gran cantidad de fallos (como se puede ver en el capítulo 1), con esta técnica se pudieran inyectar más tipos de fallos que con el resto de técnicas utilizadas en el proyecto *FIT*, además de poder inyectar fallos en cualquier lugar del modelo. Los resultados de los experimentos llevados a cabo con la inyección de fallos basada en VHDL se explicarán en la segunda parte de este capítulo.

7.3.2.6 Inyección de fallos basada en C-SIM

C-Sim es una herramienta de programación para la simulación de procesos discretos. Es una extensión del leguaje C obtenido mediante la inclusión de funciones y *macros* basadas en *SIMULA*. *C-Sim* es una biblioteca especial de *C-funciones* y *C-macros* que es utilizada por el usuario. Este escribe su propio programa en lenguaje C el cual realiza llamadas a esta biblioteca. El área típica de aplicación de *C-Sim* es la validación funcional de programas y sistemas distribuidos, paralelos y tolerantes a fallos [Hlavièka96].

El propósito del *C-Sim* dentro del proyecto *FIT* es implementar una descripción funcional del protocolo *TTP/C* precisa, portable e independiente del integrado real, basada en la especificación del protocolo. Esta descripción será incorporada en el entorno de simulación *C-Sim* con el fin de obtener un modelo ejecutable de la actividad de un *cluster TTP*. Utilizando este modelo es posible verificar las propiedades del protocolo *TTP* mediante la simulación de la inyección de fallos. Los fallos se inyectan corrompiendo las estructuras de datos del modelo, como pueden ser las áreas de memoria. El modelo *C-Sim* se puede aplicar principalmente en dos niveles: especificación y aplicación [Herout02].

7.3.2.7 Comparación de las técnicas inyección de fallos

Dentro de los objetivos del proyecto *FIT*, uno de ellos era la "*comparación de la efectividad de las diferentes técnicas de inyección de fallos para la evaluación de la tolerancia a fallos*". Así pues, a medida que se iba desarrollando el proyecto se iban viendo las diferentes características de cada técnica. La Tabla 25 muestra un resumen de la comparación de las diferentes técnicas de inyección [FIT02d] utilizadas durante el proyecto *FIT*.

Técnica de Inyección	Ventajas	Desventajas
Inyección de fallos a nivel de *pin*	▪ Representatividad de los fallos (se inyectan fallos reales). ▪ No produce ninguna sobrecarga en el sistema bajo prueba. ▪ Se puede inyectar un gran número de fallos en un tiempo relativamente corto. ▪ Es relativamente sencillo automatizar la ejecución de los experimentos de inyección.	▪ Elevado coste de desarrollo, tanto técnico como temporal. ▪ Observabilidad restringida a eventos externos. ▪ La cobertura está limitada a las señales de entrada/salida.
Inyección de fallos mediante iones pesados	▪ Representatividad de los fallos (se inyectan fallos reales). ▪ Se puede inyectar un gran número de fallos en un tiempo relativamente corto. ▪ No produce ninguna sobrecarga en el sistema bajo prueba. ▪ Inyecta fallos en cualquier lugar interno del sistema bajo prueba.	▪ Riesgo de destrucción del sistema inyectado. ▪ Potencialmente peligrosa para la salud. ▪ Poca controlabilidad del lugar de inyección. ▪ El *hardware* y la fuente de iones pesados para realizar la inyección son caros.
Inyección de fallos implementada por *software*	▪ Gran controlabilidad espacial y temporal a la hora de inyectar el fallo. ▪ Gran precisión en la medición de parámetros temporales. ▪ Es relativamente sencillo automatizar la ejecución de los experimentos de inyección. ▪ No necesita *hardware* adicional para realizar la inyección.	▪ Dificultad para emular fallos permanentes. ▪ Introduce sobrecarga en el sistema. ▪ No es sencillo reutilizar el mismo sistema de inyección en otro sistema.
Inyección de fallos basada en el microcódigo	▪ No introduce sobrecarga en el sistema. ▪ Gran reproducibilidad de los experimentos. ▪ Precisión en la selección de la localización de la inyección.	▪ Sólo inyecta fallos permanentes. ▪ Se necesita *hardware* específico para realizar la inyección. ▪ El coste del desarrollo (técnico y temporal) de la herramienta de inyección es elevado. ▪ No es sencillo reutilizar el mismo sistema de inyección en otro sistema.
Inyección de fallos basada en VHDL	▪ No introduce sobrecarga en el sistema. ▪ Precisión total a la hora de inyectar fallos simples o múltiples. ▪ Reproducibilidad de los experimentos. ▪ Se puede observar cualquier componente del modelo. ▪ La herramienta de inyección se puede utilizar con cualquier modelo VHDL.	▪ Elevado coste temporal de las simulaciones. ▪ El coste del desarrollo de la herramienta de inyección es elevado.
Inyección de fallos basada en C-Sim	▪ Se puede observar cualquier componente del modelo. ▪ Flexibilidad de la herramienta a la hora de inyectar fallos, observar el sistema, etc. ▪ Experimentos totalmente deterministas.	▪ Simulación discreta del tiempo. ▪ Independiente del *hardware* final (el modelo está construido basándose en las especificaciones).

Tabla 25. Principales ventajas y desventajas de las diferentes técnicas de inyección de fallos observadas durante el proyecto FIT [FIT02d].

7.4 Validación de la arquitectura Time-Triggered mediante inyección de fallos en VHDL

Para finalizar este capítulo, en este bloque se van a explicar detalladamente los diferentes experimentos de inyección llevados a cabo en los dos modelos en VHDL del *TTP/C*: el C1 y el C2, que como se ha comentado anteriormente, se han efectuado durante el proyecto *FIT*. Hay que mencionar que, inicialmente, en el proyecto *FIT* sólo estaba previsto el estudio del controlador de comunicaciones del *TTP/C-C*1. Sin embargo, los resultados obtenidos por las diferentes técnicas de inyección, así como la disponibilidad del modelo del *TTP/C-C2*, permitieron realizar algunos experimentos de inyección sobre este último modelo.

Los diferentes experimentos de inyección realizados se pueden dividir en dos grupos:

1. Validación del modelo en VHDL, mediante el estudio del funcionamiento de los mecanismos de detección de errores del mismo en presencia de fallos transitorios.
2. Comparación y/o combinación de la inyección de fallos basada en VHDL con la inyección de fallos a nivel de *pin*.

7.4.1 Inyección de fallos en los modelos VHDL de los controladores TTP/C-C1 y TTP/C-C2

En este apartado se van a describir los experimentos de inyección realizados con las dos versiones del controlador *TTP/C*: el *TTP/C-C1* y el *TTP/C-C2*. En primer lugar, se describirán los experimentos de ajuste que se hicieron, para después detallar los experimentos de validación de ambos controladores (*TTP/C-C1* y *TTP/C-C2*). Para finalizar, se explicarán los experimentos de inyección realizados durante la colaboración con la inyección de fallos a nivel de *pin*.

7.4.1.1 Experimentos de ajuste en el TTP/C-C1

Antes de empezar con las inyecciones masivas, se realizó un estudio teórico del modelo, con el fin de determinar aquellos puntos que pudieran ser más susceptibles de provocar un comportamiento erróneo en el controlador, así como poder identificar parámetros necesarios para la realización de los diferentes experimentos de inyección. Los pasos seguidos durante esta fase fueron:

1. **Definición de los objetivos de los experimentos de inyección de fallos**. De acuerdo con los objetivos originales marcados por el proyecto *FIT* [Madritsch01], el objetivo de los experimentos de inyección es la verificación del comportamiento de silencio ante fallos del controlador *TTP/C* en presencia de fallos transitorios.
2. **Identificación de los puntos de inyección**. Como se ha visto en el capítulo 1, VFIT permite la selección de cualquier elemento del modelo en VHDL para realizar la inyección de fallos. Después de estudiar el modelo del controlador *TTP/C*, se decidió que los mejores lugares para realizar la inyección de fallos eran:
   - La Unidad de Control del Protocolo (PCU).
   - El *bus* de comunicaciones.
   - El Guardián del *Bus*.
   - La Unidad de Control Temporal (TCU).

   Obviamente, a medida que avanzó el proyecto *FIT*, se identificaron nuevas localizaciones en las que realizar campañas de inyección de fallos, e incluso, se eliminaron algunas de las localizaciones identificadas en este primer paso.

3. **Señales a observar**. Para poder comprobar el resultado de las inyecciones de fallos, es necesario identificar aquellas señales que nos proporcionarán toda la información necesaria del comportamiento del sistema ante fallos. En este caso, se pueden resumir en:
   - *Bus* de comunicaciones.
   - Señales de recepción y transmisión y sus señales de habilitación.
   - Señales de error (o que indican la ocurrencia de un funcionamiento erróneo).
   - Señales de habilitación del funcionamiento del controlador de comunicaciones.
4. **Cláusulas**. Para determinar el correcto funcionamiento del controlador *TTP/C*, son necesarias unas condiciones especiales que denominamos cláusulas. Estas cláusulas permiten conocer el comportamiento del sistema mediante expresiones *booleanas* en las cuales se comprueba el valor de diferentes señales, tal como se ha visto en "*señales a observar*". VFIT maneja diferentes tipos de cláusulas, relacionadas con eventos a controlar. En los experimentos realizados, se han utilizado básicamente dos tipos de cláusulas: cláusulas de detección y cláusulas de finalización correcta. El primer grupo de cláusulas (cláusulas de detección) se puede definir como una condición especial que se tiene que cumplir para poder asegurar que el modelo ha detectado un error (p.ej. un bit de interrupción puesto a '1'). Las cláusulas de finalización correcta se definen como una condición especial que se tiene que cumplir para poder asegurar que el modelo ha finalizado correctamente la simulación. Estas cláusulas pueden controlar el valor de una o varias señales. Además, el examen de estos valores puede ser dependiente de un instante de tiempo dado, el cual también se puede controlar.
5. **Tiempo de simulación**. El tiempo de simulación del modelo puede ser dividido en dos partes. La primera es el período de tiempo en el que la inyección es efectiva, es decir, es el período de tiempo durante el cual el fallo inyectado afecta al sistema. La segunda parte es el período de recuperación del controlador. Según las especificaciones del *TTP/C* [TTP/C99], un sistema *TTP/C* debe recuperarse en una ronda de TDMA. Debido a esta razón, se ha elegido el primer ciclo de *cluster* para realizar la inyección, y el segundo ciclo para comprobar si el controlador se ha recuperado (ver la Figura 48).
6. **Resultados de la inyección de fallos**. Según [FIT01a], podemos definir una avería como una violación del silencio ante fallos. Esta definición se puede dividir en dos:
   - Se puede asumir que el nodo presenta una violación del silencio ante fallos en el dominio de los valores si el controlador está funcionando correctamente, pero está enviando mensajes con valores erróneos.
   - En el caso de que dos nodos pierdan simultáneamente su pertenencia al *cluster*, se puede asumir que se ha producido una violación del silencio ante fallos en el dominio temporal.

A partir de este estudio teórico previo, se llevaron a cabo un grupo de experimentos con el fin de afinar o ajustar tanto los parámetros de inyección de los siguientes experimentos como el funcionamiento de VFIT. Debido a esta razón, se realizaron menos inyecciones que en el resto de campañas. Los parámetros de los tres experimentos de ajuste que se realizaron se muestran a continuación:

- **Número de inyecciones**: 500 por experimento.
- **Lugar de la inyección**:

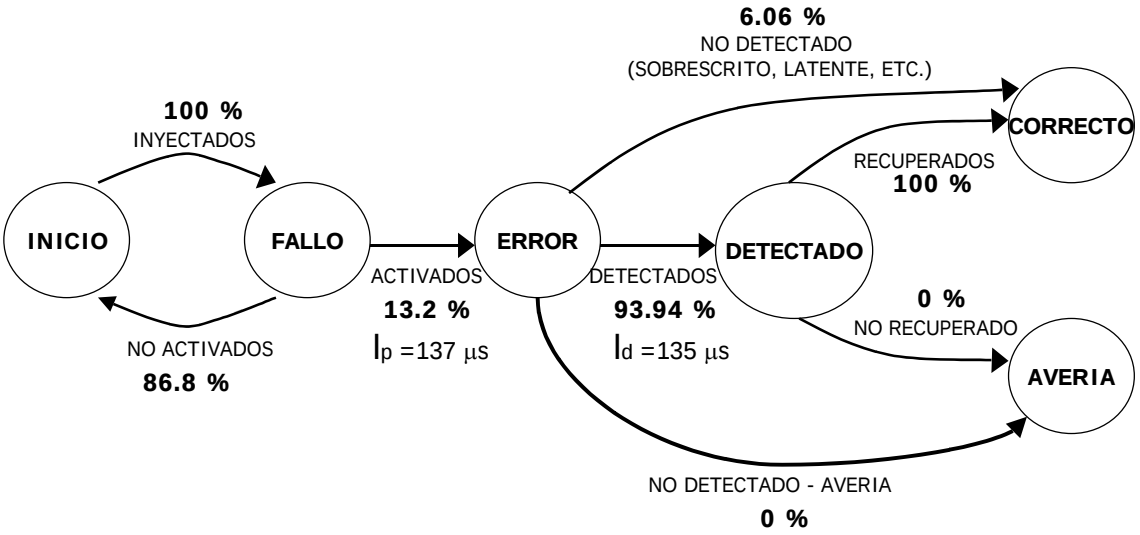
  Exp.1: Todas las señales y variables del Guardián del *Bus*.

  Exp.2: Señal de control temporal del Guardián del *Bus*.

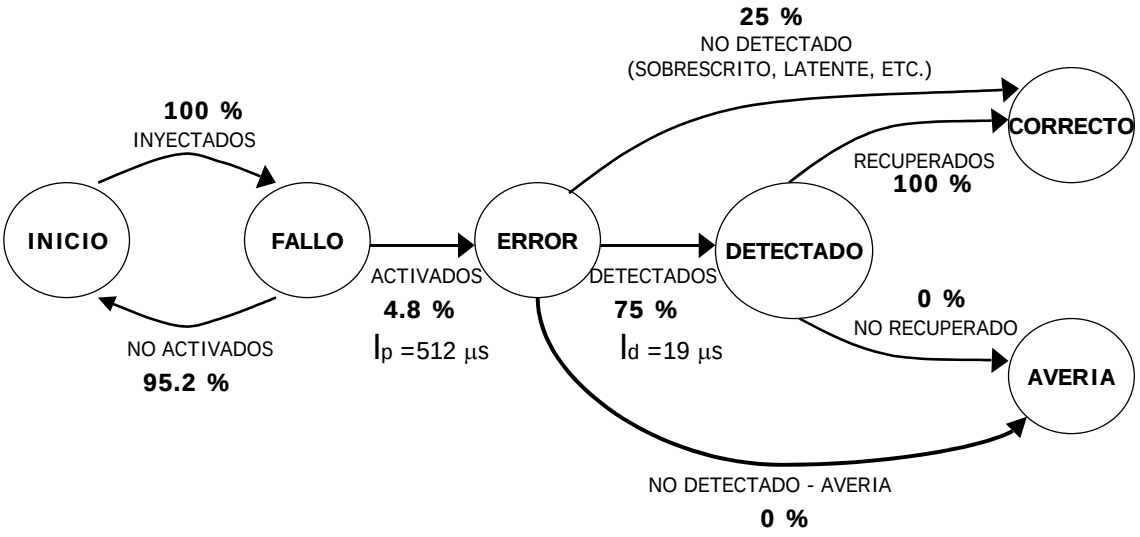
  Exp.3: Señales externas del controlador, es decir, aquellas señales que formarán la interfaz con el exterior una vez sintetizado el circuito, pero que no pertenecen al grupo de "*señales a observar*".

- **Tipos de fallos**: *Pulse* (en lógica combinacional), *Bit-flip* (en registros y biestables) e Indeterminación.
- **Instante de inyección**: Seleccionado aleatoriamente durante el primer ciclo de *cluster*.
- **Duración de los fallos**: En los tres experimentos se inyectaron fallos transitorios de una duración lo suficientemente larga como para asegurar la activación de los mismos. En el primer experimento (todas las señales y variables del Guardián del *Bus*) los fallos tenían una duración aleatoria dentro del rango [30.0T, 40.0T], donde T es la duración del ciclo de reloj. En el segundo experimento, la duración estaba en el rango [50.0T, 60.0T] y en el tercero, la duración del fallo se seleccionó en el rango [½ periodo de SRU, 1 periodo de SRU].

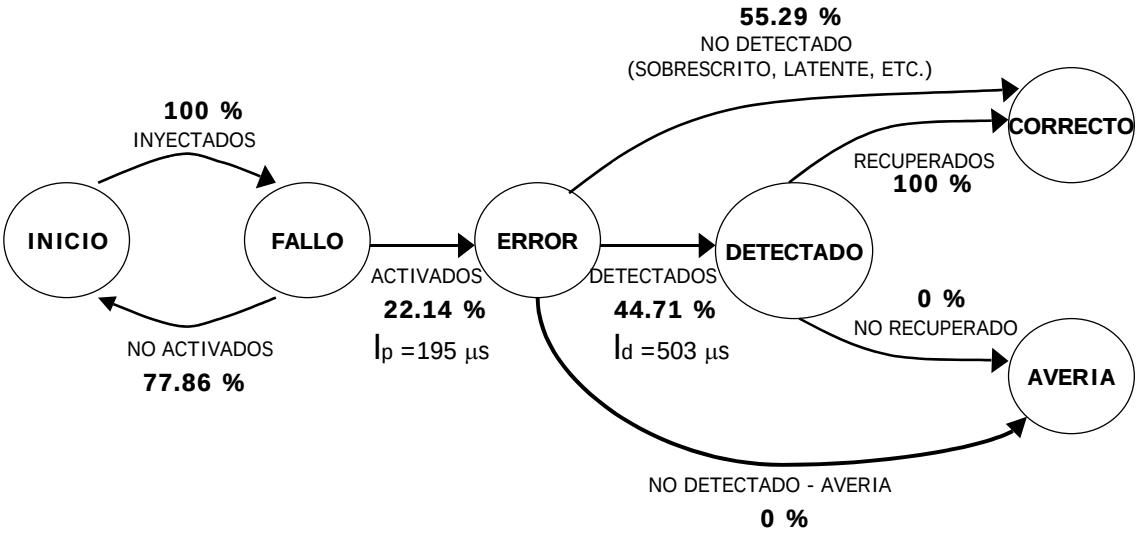
Aunque en estos experimentos, los resultados no son lo más importante, en la Figura 56 se pueden ver los *grafos de predicados de los mecanismos de tolerancia a fallos*. Estos grafos serán los que se utilizarán en los siguientes experimentos para representar los resultados. A pesar de no encontrar ninguna avería (o violación del silencio ante fallos), estos experimentos confirmaron el ajuste de la herramienta así como de los diferentes parámetros a observar en los siguientes experimentos.



(a)



(b)



(c)

Figura 56. Grafo de predicados de los mecanismos de tolerancia a fallos de los experimentos de ajuste. (a) Primer experimento. (b) Segundo experimento. (c) Tercer experimento.

De todas maneras, se pueden obtener algunas conclusiones a partir de los resultados de estos experimentos. En los dos primeros, el porcentaje de errores activados es bastante bajo, sobre todo en el segundo experimento. Esto puede deberse tanto al pequeño número de fallos inyectados como a la duración del fallo, que puede ser demasiado corta. Este porcentaje ha aumentado en la tercera campaña, en la que la duración de los fallos es mayor. Las latencias de propagación (l_p) son muy variables. Esto puede deberse a:

- Se activan pocos fallos, sobretodo en el segundo experimento.
- Respecto al segundo experimento, la señal inyectada se actualiza constantemente, con lo que el fallo inyectado es sobrescrito. Es decir, el periodo de tiempo durante el cual el fallo está activo es muy pequeño, con lo que el intervalo que pasa desde la inyección del fallo hasta que esta señal es tenida en cuenta por el controlador suele ser grande.

El mayor porcentaje de fallos detectados se presenta en el primer experimento aunque la latencia de detección (l_d) es mucho menor en el segundo experimento. La señal inyectada en el segundo experimento es utilizada por el modelo para la temporización del Guardián del *Bus*. Una variación en esta señal implica una variación en la temporización del controlador, la cual se detecta rápidamente. Sin embargo, en bastantes ocasiones, el controlador es capaz de tratar este tipo de fallo sin activar ningún mecanismo de detección de errores, y producir un resultado correcto (es decir, la carga de trabajo finaliza correctamente), tal y como se puede comprobar en la transición *No Detectado (Sobrescrito, Latente, etc.)* que es mucho mayor en el segundo experimento que en el primero. Esta transición refleja la *Redundancia Intrínseca* del sistema.

Mención aparte merece el tercer experimento. En este caso las señales inyectadas pertenecen a la interfaz externa. El porcentaje de fallos activados es mayor que en los otros dos experimentos debido principalmente a la mayor duración de los fallos inyectados. Sin embargo, se puede ver que la latencia de detección es la mayor. Es decir, que hasta que el fallo llega a una de las señales de detección de errores, pasa un tiempo mayor que en los otros experimentos. A pesar de este dato, el porcentaje de fallos detectados es el menor de los tres experimentos. De este hecho podemos concluir que las señales seleccionadas no afectan al sistema de una forma grave, aunque una vez que el fallo se ha activado, a los mecanismos de detección de errores del sistema les cuesta detectar el error. Como se puede ver en la Figura 56-c, una gran parte de los errores son detectados por la *Redundancia Intrínseca* del sistema.

En ningún grafo se ven las latencias de recuperación. Esto es debido a que el modelo del controlador no tiene ningún mecanismo de recuperación implementado, debido a dos razones: i)

Estos mecanismos están implementados en el *host*, y nosotros lo que estamos validando es el controlador de comunicaciones, y ii) el modelo de *host* utilizado es muy simple. Básicamente, es un proceso que realiza operaciones elementales de mantenimiento con el fin de indicar que el *host* está funcionando correctamente. En especial, ejecuta el algoritmo de la señal de vida del *host* (del inglés, *host life-sign algorithm*). Este algoritmo indica al controlador que el *host* está activo.

7.4.1.2 Validación del controlador de comunicaciones

Una vez realizados los experimentos de ajuste, se pasó a validar el controlador de comunicaciones. Durante esta validación se encontró un error de diseño en el controlador de comunicaciones del *TTP/C-C1*. Concretamente, se encontró un error en la implementación del algoritmo del *clique avoidance*, tal y como se explica a continuación.

7.4.1.2.1 Error de diseño en el TTP/C-C1: algoritmo del clique avoidance

El algoritmo de *clique avoidance* detecta fallos que están fuera de las hipótesis de fallos y que son intolerables en el nivel de protocolo [Maier02]. Como se comentó en el capítulo anterior, el mecanismo de *clique avoidance* detecta fallos en múltiples componentes e inconsistencias, así como soporta la estrategia NGU (del inglés *Never-Give-Up*). En los experimentos que se presentan a continuación [Gracia02b, Gracia03a, Gracia03b], se han encontrado fallos en la implementación de este algoritmo, no en su definición.

Este error se detectó al realizar una validación del modelo del *TTP/C-C1*. El objetivo de estos experimentos era comprobar si el modelo observaba la condición de silencio ante fallos, es decir, un nodo funciona correctamente o no transmite nada. Además, también se comprobaría el funcionamiento de los mecanismos de detección de errores implementados en el modelo VHDL y su efectividad frente a fallos transitorios, tal y como marcaban las especificaciones del proyecto *FIT*. Antes de continuar, conviene explicar cuáles son los mecanismos de detección de errores implementados en el modelo, que son los siguientes [TTP/C99]:

- Error del *Host* (en inglés, *Host Error* o **HE**). Este mecanismo detecta errores en el host. Cuando esto ocurre, el controlador pasa al estado *pasivo*, observando un comportamiento de silencio ante fallos. Si el error es transitorio, el *host* puede reactivar al controlador actualizando el campo de la señal de vida del *host*. Este error puede ser producido por:
  - ⇨ Error de Violación de Modo. El *host* hace una petición de cambio de modo el cual no está permitido hacer.
  - ⇨ Error de Período Ocupado (en inglés, *Slot Occupied Error*). El período de envío está ocupado por otro controlador.
  - ⇨ Error del Protocolo NBW[52]. El NBW ha detectado un error.
- Error de Protocolo (en inglés, *Protocol Error* o **PE**). Detecta errores internos del protocolo *TTP/C*. En este caso, se suspende la ejecución del protocolo hasta que el *host* active de nuevo al controlador. Este error puede ser provocado por varias causas:
  - ⇨ Error de Reconocimiento (en inglés, *Acknowledgement Error*). El controlador está en desacuerdo con la mayoría del *cluster*.
  - ⇨ Error de Pertenencia (en inglés, *Membership Error*). Se ha alcanzado el máximo número de fallos de pertenencia sucesivos especificado en el contador correspondiente.

[52] NBW: del inglés *Non-Blocking Write*. Este protocolo evita que el controlador y el host realicen una lectura y una escritura a la vez. Es decir, cuando uno de los dos está escribiendo en memoria, el otro debería esperar para leer el valor actualizado. Si esto no ocurre, este mecanismo detecta este error.

- ⇨ Apagón del Sistema de Comunicaciones (en inglés, *Communication System Blackout*). Solamente se ha detectado nuestra actividad en el *bus* durante la última ronda de TDMA.
- ⇨ Error de Sincronización (en inglés, *Synchronisation Error*). El subsistema de sincronización del reloj ha detectado un error.
- ⇨ Error del Guardián del *Bus* (en inglés, *Bus Guardian Error*). El Guardián del *Bus* ha detectado un error transitorio o permanente.
- ⇨ Error de CRC en la MEDL (en inglés, *MEDL CRC Error*). El valor del CRC calculado para la entrada actual de la MEDL no concuerda con el CRC almacenado.
- ⇨ Descarga Completada (en inglés, *Download Completed*). El controlador ha completado su operación de descarga.

- Pérdida de Pertenencia (en inglés, *Membership Loss* o **ML**). Detecta la pérdida de pertenencia del controlador en el *cluster*.
- Error de los Mecanismos Internos Autocomprobantes (en inglés, *Built-in Self-Test Mechanisms* o **BE**). Este mecanismo está compuesto realmente por un grupo de mecanismos que detectan errores internos en el controlador. En caso de error, la ejecución del protocolo se suspende hasta que el *host* active de nuevo al controlador. Los mecanismos internos implementados en el controlador son [Steininger99]:
  - ⇨ Acuerdo en el *estado-C* (en inglés, *C-state agreement*). Cada controlador tiene un *estado-C* local, y una especie de votación que forma un protocolo para determinar el *estado-C* global. El *estado-C* consiste en tres campos: la posición de la MEDL, el tiempo global en el comienzo del periodo de TDMA actual y el vector de pertenencia. Para forzar el acuerdo sobre el *estado-C* en el sistema, el emisor calcula el CRC del mensaje junto con su *estado-C* local.

    El emisor envía su mensaje, junto con el CRC, a los otros nodos. Los receptores calculan el CRC del mensaje recibido y de sus *estado-C* locales. Si la comprobación del CRC falla, puede deberse a dos causas: el mensaje estaba corrupto durante la transmisión o existe un desacuerdo entre el *estado-C* del emisor y el *estado-C* del receptor. En ambos casos, el receptor marca el mensaje como no válido. Además, el receptor marca en el vector de pertenencia que el emisor no pertenece al *cluster* (ya que ha enviado una trama incorrecta), modificando el *estado-C*. Esta acción es considerada por el emisor como un reconocimiento negativo. El controlador de comunicaciones no enviará nada después de recibir este reconocimiento negativo.
  - ⇨ Señal de Vida del *Host* (en inglés, *Host Life Sign*). Este mecanismo habilita la comunicación entre el *host* y el controlador de comunicaciones. Es comprobado una vez por cada ronda de TDMA. Si el *host* no es capaz de actualizar este campo, el controlador de comunicaciones no enviará nada.
  - ⇨ CRC de los datos (en inglés, *End-to-end CRC*). Durante la transmisión de datos entre las diferentes tareas de la aplicación, éstos son protegidos mediante el cálculo de un CRC para cada mensaje que se envía entre cada tarea de cada nodo. El controlador de comunicaciones considera este CRC como parte del mensaje.
  - ⇨ Guardián del *Bus*. Permite la transmisión de tramas en instantes predefinidos, protegiendo la comunicación de nodos con fallos. Utiliza un reloj particular para evitar fallos en modo común con el reloj del sistema.
  - ⇨ Comprobación del CRC de la MEDL. El controlador de comunicaciones también utiliza la unidad de CRC para validar las entradas de la MEDL.

Respecto al modelo en VHDL del controlador, éste presenta una arquitectura híbrida. Durante los experimentos de inyección, se ha simulado un *cluster* completo con 6 nodos. Cada nodo está compuesto por un modelo comportamental de una EPROM (en la cual está

almacenado el código del protocolo), un *transceiver*, un *host*, y el modelo sintetizable del controlador *TTP/C-C1*, así como otros bloques funcionales de bajo nivel, tal y como se puede ver en la Figura 57.

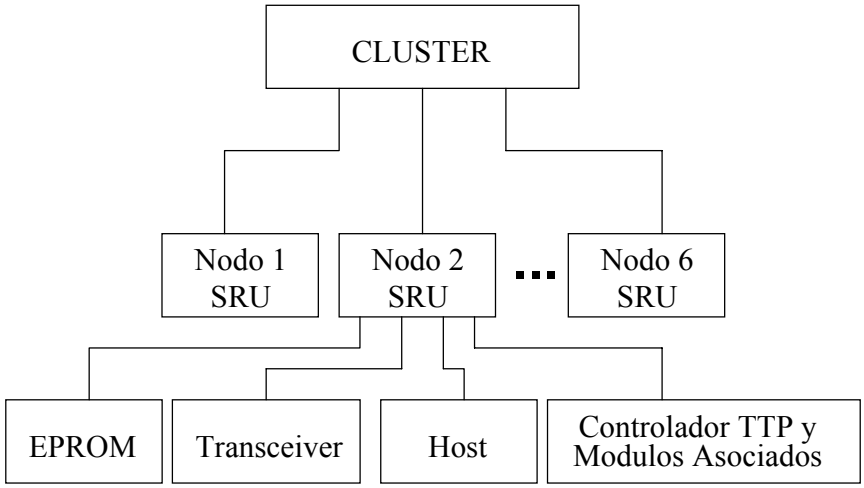


Figura 57. Diagrama de bloques del sistema simulado [Gracia03b].

Objetivos y parámetros de los experimentos de inyección de fallos

Como se dijo anteriormente, el principal objetivo de los experimentos de inyección de fallos realizados fue comprobar si el controlador *TTP/C-C1* cumple la condición de silencio ante fallos en presencia de fallos transitorios. Para conseguir este objetivo, se ha comprobado el funcionamiento del protocolo *TTP/C*. Las hipótesis de fallos de este protocolo se pueden resumir en dos:

- La condición de silencio ante fallos: el nodo envía resultados correctos o no envía nada.
- Un fallo simple no perturbará el funcionamiento del resto del sistema.

La carga de trabajo utilizada es una aplicación de prueba que ejercita todos los módulos y sistemas del controlador. Las condiciones para los diferentes experimentos han sido:

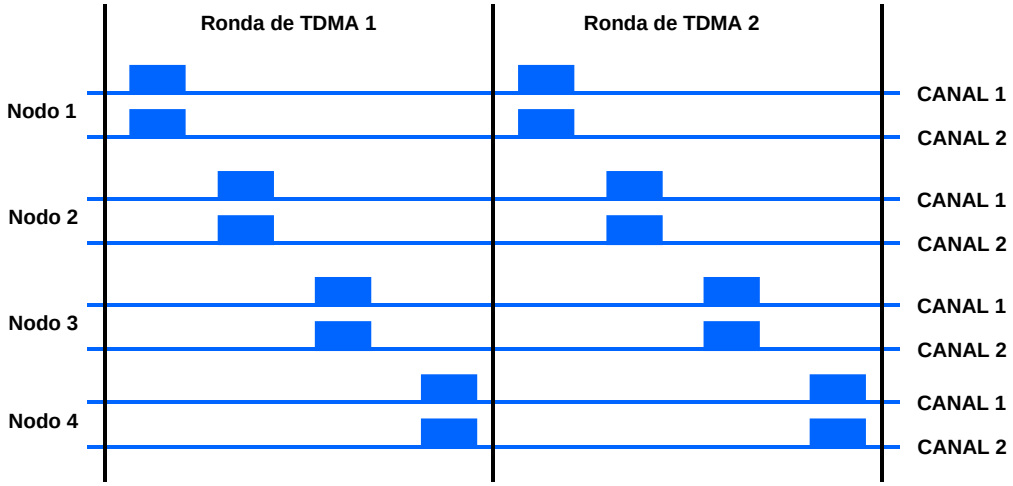
- **Lugar de la inyección**. Todas las señales y variables de la Unidad de Control del Protocolo (PCU), de la Unidad del cálculo del CRC y de la Unidad de Control Temporal (TCU), además del Registro de Instrucciones de la PCU (IR-PCU). Estos módulos han sido seleccionados por su importancia y su función crítica en el sistema.
- **Número de inyecciones**. 3000 en la PCU y en el IR-PCU, y 2000 en el CRC y el TCU.
- **Instante de la inyección**. Seleccionado aleatoriamente (utilizando una distribución uniforme) durante el primer ciclo de *cluster* (tal como se puede ver en la Figura 48).
- **Duración de los fallos**. Transitorios, con una duración aleatoria seleccionada en el rango [½ periodo de SRU, 1 periodo de SRU].
- **Distribución de los fallos**. Uniforme.
- **Modelos de fallos**. *Bit-flip* (en elementos de memoria), *Pulse* (en lógica combinacional), Indeterminación y *Delay*.

Resultados

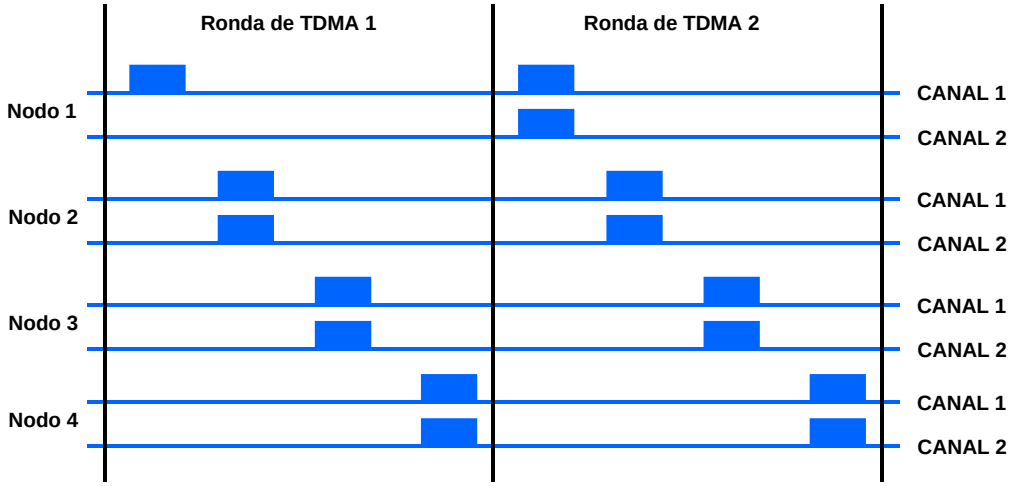
En primer lugar, hemos definido una avería como una violación del silencio ante fallos. Con esta premisa, se puede decir que se han encontrado varias averías, las cuales causan un bloqueo del envío de mensajes, es decir, ningún nodo del sistema envía mensajes, violando las hipótesis de fallos del protocolo de comunicaciones *TTP/C*. Estas averías han sido provocadas por una configuración especial de la carga de trabajo ejecutada en el modelo del *host* (hay que recordar

que tanto la carga de trabajo como el modelo del *host* son muy simples, implementados únicamente para probar el modelo del controlador de comunicaciones).

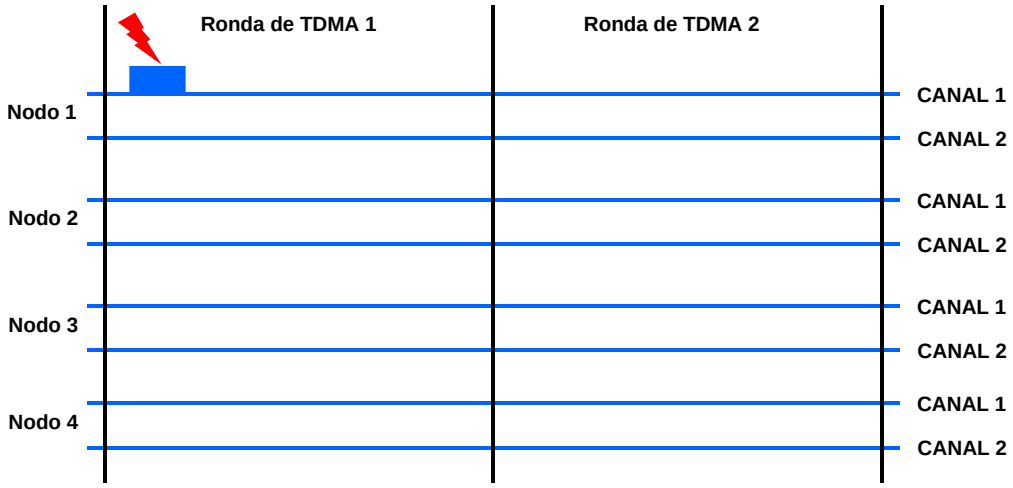
La carga de trabajo está configurada de tal manera que únicamente envía un mensaje por uno de los canales durante el primer periodo de envío o periodo de SRU, en vez de enviar dos mensajes usando los dos canales de transmisión, tal y como ocurre durante el resto de la ejecución de la carga de trabajo. La Figura 58-a muestra la configuración normal, en la que todos los mensajes se envían replicados utilizando ambos canales. La Figura 58-b muestra la configuración especial de la carga de trabajo utilizada en los experimentos de inyección, en la que sólo se envía un mensaje por un canal durante el primer periodo de SRU. Si el fallo inyectado provoca que este primer mensaje no llegue de forma correcta, es decir, el resto de nodos recibe un mensaje corrompido, todos los nodos marcan al primero como un nodo no válido. El efecto que se produce es el mostrado en la Figura 58-c: ningún nodo del sistema envía más mensajes, violando las hipótesis de fallos del protocolo de comunicaciones.



(a)



(b)



(c)

Figura 58. Avería del algoritmo del *clique avoidance*. a) Configuración normal. b) Configuración utilizada en los experimentos de inyección. c) Avería producida.

La avería está causada por una implementación errónea del algoritmo del *clique avoidance*. Según la especificación del protocolo *TTP/C* [TTP/C99], durante la inicialización del sistema (en terminología *TTP* se denomina *start-up*), todos los nodos del *cluster* se integran después de recibir un mensaje especial (conocido como *Cold-Start Frame*). Después de recibir este mensaje, el contador de mensajes correctos recibidos en la última ronda de TDMA (en el protocolo este contador se denomina *Accept Counter*) de todos los nodos es puesto a 1. En nuestro caso, como la transmisión del primer mensaje no es correcta, el contador de mensajes recibidos semánticamente incorrectos (en el protocolo este contador se denomina *Failed Counter*) es puesto a 1.

Antes de su periodo de envío, todos los nodos realizan una comprobación del *clique avoidance*. El algoritmo de *clique avoidance* comprueba si en la última ronda de TDMA el número de mensajes recibidos correctamente (*Accept Counter*) es mayor que la suma de los mensajes semánticamente incorrectos (*Failed* Counter) y no válidos (*Invalid Counter*), siguiendo el algoritmo de la Figura 59. Así pues, cuando el nodo 2 realiza la comprobación del *clique avoidance*, ésta falla porque el *Accept Counter* no es mayor que la suma de *Failed Counter* e *Invalid Counte*r, con lo que el nodo 2 entra en el estado *congelado*, pasando lo mismo con el resto de nodos del sistema, con lo que un fallo en un nodo se ha propagado al resto del sistema. La solución de este error es muy sencilla. Basta con inicializar el contador *Accept Counter* a 2 en vez de a 1 [FIT02a].

```
if Accept Counter > (Failed Counter + Invalid Counter)
then
reset (Accept, Failed and Invalid Counter)
else
goto freeze_state
fi
```

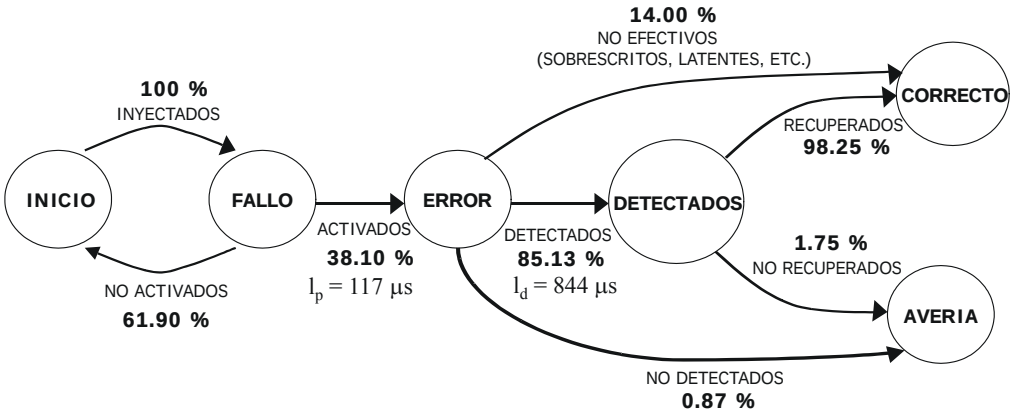
Figura 59. Algoritmo del *clique avoidance* [TTP/C99].

A modo de resumen, en la Tabla 26 se puede ver el porcentaje de averías respecto de los fallos activados para los distintos experimentos.

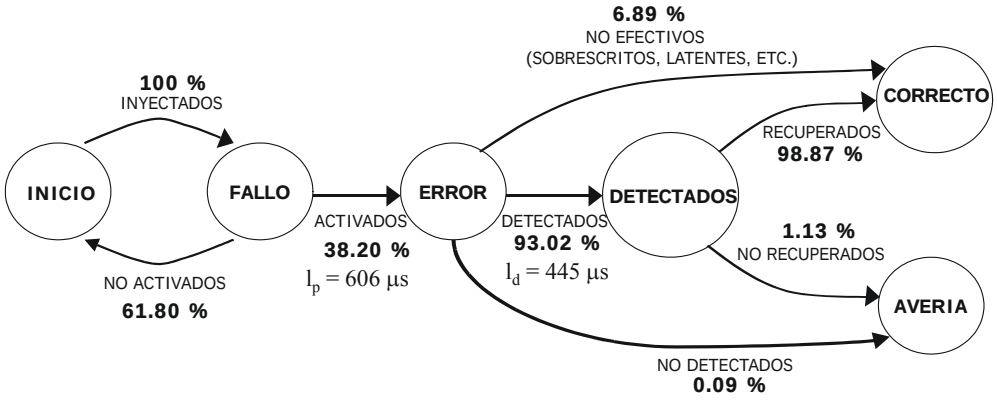
	IR-PCU	PCU	CRC	TCU
CORRECTO	97.64 %	98.86 %	97.44 %	98.72 %
AVERIA	2.36 %	1.14 %	2.56 %	1.28 %

Tabla 26. Porcentaje de averías respecto de los fallos activados [Gracia03b].

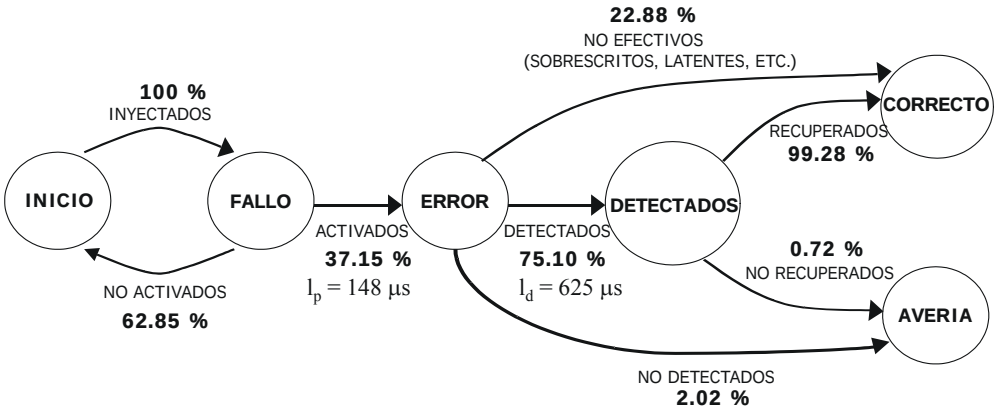
Hay que recordar, sin embargo, que estos experimentos también tenían como objetivo comprobar el funcionamiento de los distintos mecanismos de detección de errores. Como se ha comentado anteriormente, para ver la evolución de los fallos en el sistema, se utiliza el *grafo de predicados de los mecanismos de tolerancia a fallos*. La Figura 60 muestra estos grafos para los experimentos del IR-PCU, PCU, CRC y TCU [Gracia02b, Gracia03a, Gracia03b].



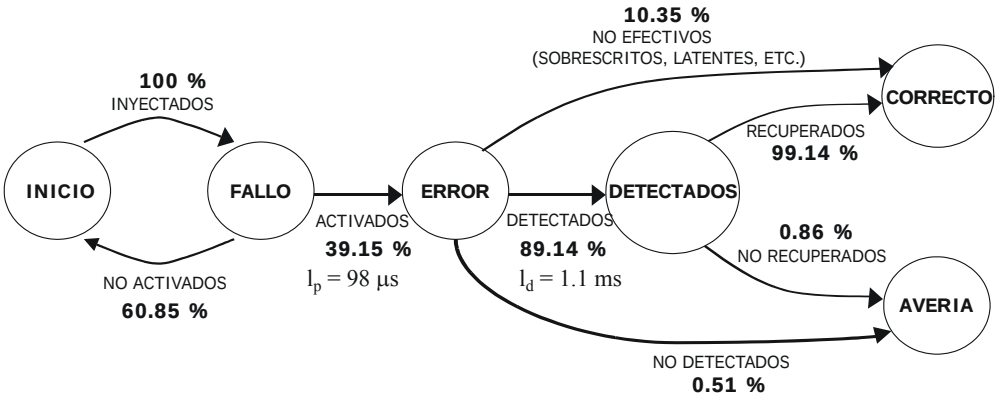
(a)



(b)



(c)



(d)

Figura 60. Grafos de predicados de los mecanismos de tolerancia a fallos [Gracia03b]. Inyección en: (a) IR-PCU. (b) PCU. (c) CRC. (d) TCU.

Las principales conclusiones que se pueden obtener de estos grafos son:

- Los porcentajes de errores activados son similares en todos los experimentos. Los fallos inyectados sólo se activan cuando la ejecución del microcódigo coincide con la inyección. La mayor parte del tiempo, el microcódigo ejecuta un bucle a la espera de algún evento (recepción/transmisión de un mensaje, cálculo de algún CRC, etc.). Cuando la inyección coincide con alguna de estas operaciones, el fallo normalmente se activa.
- La inyección en el CRC conlleva el porcentaje más bajo de errores detectados, mientras que la inyección en la PCU exhibe el porcentaje más alto. Este hecho muestra que el CRC es uno de los puntos más sensibles del sistema, ya que los errores se detectan con más dificultad. La Tabla 26 confirma este hecho.
- En los grafos aparecen dos porcentajes relacionados con las averías. La transición con la etiqueta *No Detectados* muestra las averías que han sido causadas por errores que no han sido detectados por ninguno de los mecanismos de detección de errores del controlador. Comparando los valores de los cuatro grafos, podemos ver que el módulo del CRC es el que presenta el porcentaje más alto de todos los experimentos.
- Por otra parte, la transición etiquetada como *No Recuperados* presenta las averías producidas después de detectar el error. Es decir, aunque el error ha sido detectado por alguno de los mecanismos de detección de errores del controlador, ha provocado una avería (la *Redundancia Intrínseca* del sistema no ha sido capaz de tolerar este fallo). En este caso, el punto que presenta el porcentaje más alto es el IR-PCU.
- La transición etiquetada como *No efectivos* hace referencia a aquellos errores activados pero que durante el funcionamiento normal del sistema no han producido ninguna alteración en el mismo. Esto puede deberse a que el fallo haya sido sobrescrito (por ejemplo, un fallo inyectado en un registro es sobrescrito antes de que pueda afectar al sistema), o es un error latente, es decir, es un error activado y que en un futuro puede ser causante de una avería o ser sobrescrito, pero el tiempo de simulación es demasiado corto para que esto pase.
- Con esta serie de datos podemos decir que la probabilidad de averías del sistema (o violaciones del silencio ante fallos) se incrementa cuando existe un funcionamiento erróneo en el CRC o en el IR-PCU, siendo estos dos puntos los más sensibles del sistema, como se puede ver en Tabla 26.
- Respecto a las latencias, según la especificación del controlador *TTP/C-C1*, un *cluster* correctamente configurado puede tolerar cualquier fallo físico simple. Si un fallo transitorio tiene una duración menor que un período de SRU, el sistema se recuperará en una ronda de TDMA. En las campañas aquí mostradas, se han inyectado fallos cuya

duración está en el rango [½ periodo de SRU, 1 periodo de SRU]. En todos los casos (excepto en las averías, por supuesto) se ha cumplido la especificación.

- La TCU presenta la latencia de propagación más baja y la latencia de detección más alta. Es decir, los fallos inyectados en este módulo afectan antes al sistema y éste tarda más en detectarlos. Por otra parte, la PCU tiene la latencia de propagación mayor (los fallos se activan más tarde) y la latencia de detección más baja (los fallos se detectan más rápidamente).
- No se han representado las latencias de recuperación porque como se dijo anteriormente, los mecanismos de recuperación están implementados en el controlador del *host*, y el modelo que se tiene de este controlador es muy simple, además de que se está evaluando el controlador de comunicaciones.
- Las transiciones marcadas como *Recuperados* en los diferentes grafos muestran la *Redundancia Intrínseca* del sistema, que en este caso es bastante grande.

En resumen, podemos obtener dos grandes conclusiones de los diferentes experimentos:

1. Los diferentes mecanismos de detección de errores del controlador de comunicaciones presentan una gran efectividad, tal y como se puede ver en los altos porcentajes de errores detectados (Figura 60 y Tabla 26). Respecto al alto porcentaje de errores recuperados, al no haber implementados mecanismos de recuperación en el *host*, se consideran errores recuperados aquellos errores que provocan un comportamiento de silencio ante fallos en el controlador.
2. Sin embargo, aunque las tasas de averías son muy bajas, se produce un número no despreciable de violaciones del silencio ante fallos.

La Figura 61 muestra los porcentajes de errores detectados por los diferentes mecanismos de detección de errores del controlador *TTP/C-C1* en cada experimento. Como se puede ver, estos porcentajes presentan algunas diferencias. Casi todos los errores son detectados por el Error de Protocolo (PE) y los Mecanismos Internos Autocomprobantes (BE). Los errores detectados por éste último mecanismo siguen el comportamiento esperado (los fallos inyectados son detectados como fallos internos [Steininger99]). La campaña IR-PCU presenta el mayor porcentaje de errores detectados por los BE, mostrando así que las inyecciones en este punto presentan los efectos más profundos en el controlador de comunicaciones.

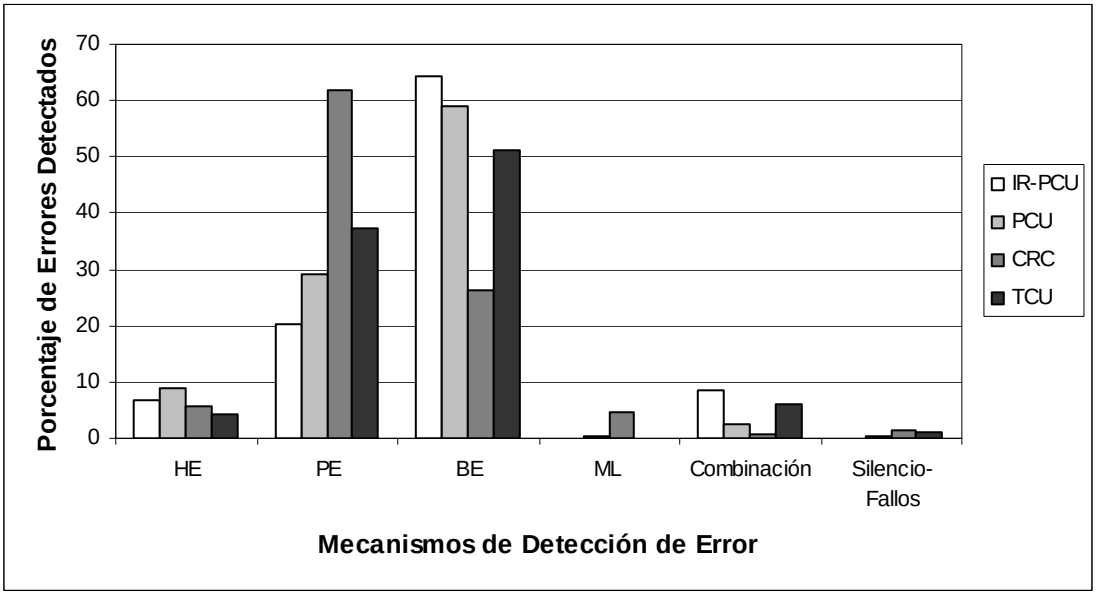


Figura 61. Porcentajes de errores detectados por los diferentes Mecanismos de Detección de Errores para todas las campañas [Gracia03b].

Por otra parte, en la campaña del CRC, la mayor parte de los errores son detectados por el Error de Protocolo (PE), es decir, están considerados como otro tipo de error interno. El mecanismo del Error de Protocolo señala un error causado por una de las siguientes causas [TTP/C99]:

- Ha ocurrido un error de CRC.
- El controlador está en desacuerdo con la mayoría del *cluster*.
- Ha habido un apagón en el sistema de comunicaciones.
- El Guardián del *Bus* ha detectado un error (permanente o transitorio).

En cualquier caso, en todas las campañas y después de detectar el error, la ejecución del protocolo se suspende hasta que el *host* vuelve a reconectar al controlador otra vez. Por otra parte, cuando los mecanismos Pérdida de Pertenencia (ML) y Error del *Host* (HE) detectan un error, provocan el comportamiento de silencio ante fallos y causan el cambio del controlador al estado *pasivo*, dejando el controlador de enviar mensajes (comportamiento normal de silencio ante fallos). HE es activado cuando el *host* no marca que está vivo.

En la Figura 61 están representados dos porcentajes adicionales. El porcentaje denominado *Combinación* se obtiene cuando dos o más mecanismos de detección de errores se activan juntos, y el porcentaje *Silencio-Fallos* se consigue cuando no se ha activado ningún mecanismo de detección de errores pero el controlador observa el comportamiento de silencio ante fallos. Aunque este porcentaje podría incluirse en la *Redundancia Intrínseca* del sistema, sin embargo, durante los diferentes experimentos se comprobó que el sistema había activado ciertos *bits* de estado que no se correspondían con ningún mecanismo de detección de errores. Es decir, el controlador detectaba un funcionamiento erróneo, dejaba de transmitir mensajes, y señalaba este comportamiento erróneo, aunque no lo señalaba de forma correcta.

Aparte de estos experimentos de inyección, se realizó uno más. En este experimento extra se inyectaron 2000 fallos transitorios en el banco de registros con una duración aleatoria seleccionada en el rango [½ periodo de SRU, 1 periodo de SRU]. El instante de inyección se eligió aleatoriamente durante la segunda ronda de TDMA. Se inyectaron fallos de los tipos Retardo, Indeterminación y *Bit-flip*.

La Figura 62 muestra el *grafo de predicados de los mecanismos de tolerancia a fallos* para este experimento. Como se puede observar, no se produjo ninguna avería (o violación del silencio ante fallos). Es más, todos los fallos activados fueron detectados, por lo que se confirma que el fallo en el algoritmo del *clique avoidance* sólo está presente durante la primera ronda de TDMA del modelo. Respecto a los experimentos explicados anteriormente, se ven ligeras diferencias. La principal es el porcentaje de fallos activados, que es menor que en los experimentos anteriores. Esto puede ser producido por:

- Los fallos inyectados han sido sobrescritos durante el funcionamiento normal del sistema.
- El tiempo de simulación es corto, por lo que los fallos inyectados no han tenido tiempo suficiente para activarse.
- Esta parte no es muy usada, por lo que el registro con los datos corrompidos no se ha utilizado, no produciendo la activación del fallo.

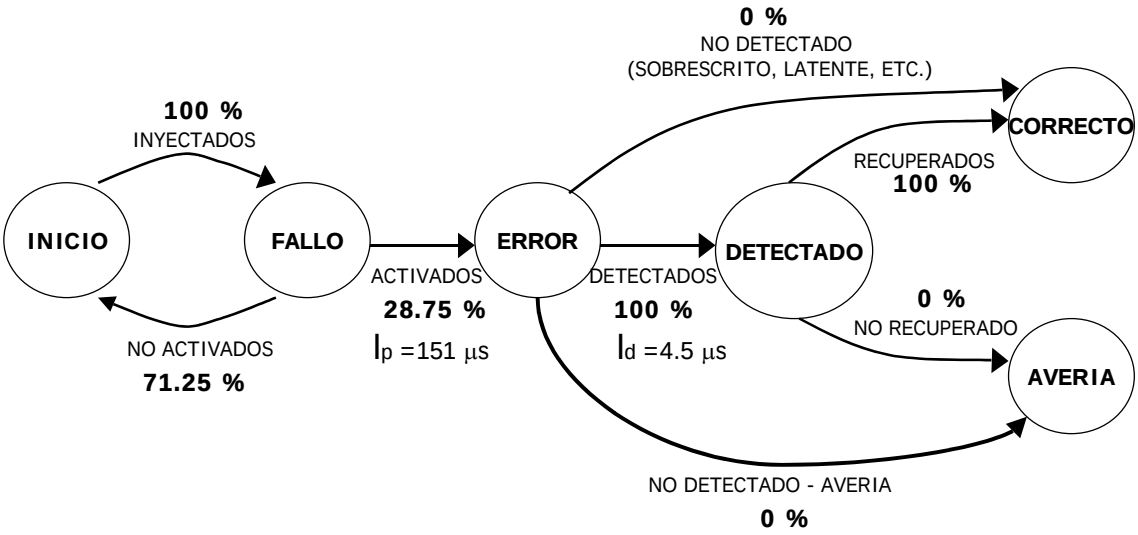


Figura 62. Grafo de predicados de los mecanismos de tolerancia a fallos para el experimento del banco de registros [FIT02a].

La Figura 63 muestra los diferentes porcentajes de errores detectados por los mecanismos de detección de errores del sistema para el experimento del banco de registros. Como se puede ver, la mayor parte de los errores se detectan mediante los Mecanismos Internos Autocomprobantes (BE) y el Error de Protocolo (PE), al igual que ocurría en los experimentos anteriores, mientras que los mecanismos Pérdida de Pertenencia (ML) y Error del *Host* (HE) no detectan ningún error. Por último, el porcentaje *Combinación* (dos o más mecanismos de detección de errores se activan juntos) detecta el resto de errores no detectados por un único mecanismo.

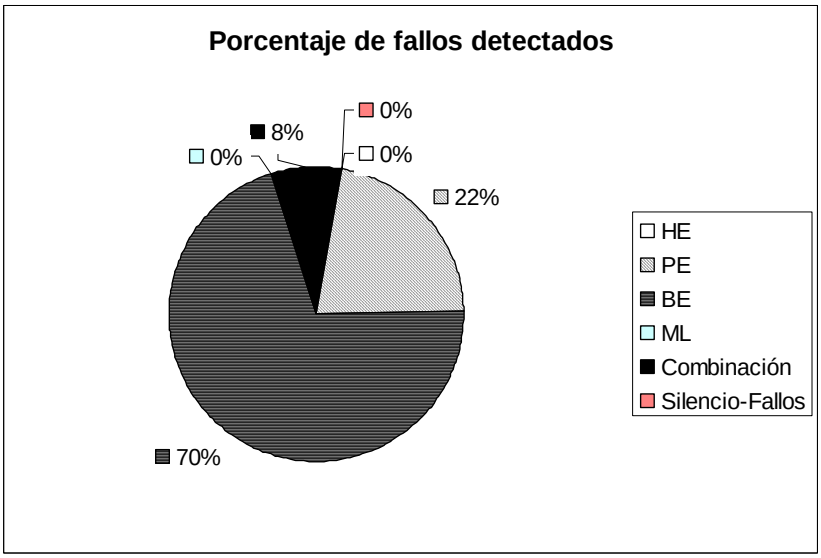


Figura 63. Porcentajes de errores detectados por los diferentes Mecanismos de Detección de Errores [FIT02a].

7.4.1.2.2 Validación del controlador TTP/C-C2

Para terminar con esta serie de experimentos, se realizaron una serie de experimentos en el modelo del controlador *TTP/C-C2* con el fin de comprobar si con estos resultados, *TTTech*[53], la compañía fabricante del controlador *TTP/C*, había corregido el error del algoritmo del *clique*

[53] http://www.tttech.com/

avoidance. Para comprobarlo, se realizaron el mismo tipo de experimentos que en el modelo del controlador *TTP/C-C1*. Los diferentes parámetros utilizados fueron:

- **Lugar de la inyección**. El Registro de Instrucciones de la PCU (IR-PCU) y todas las señales y variables de la Unidad del cálculo del CRC, de la Unidad de Control Temporal (TCU) y del Guardián del *Bus* (BG).
- **Número de inyecciones**. 2000 en el IR-PCU y en BG, y 1000 en el CRC y el TCU.
- **Instante de la inyección**. Seleccionado aleatoriamente (utilizando una distribución uniforme) durante el primer ciclo de *cluster* (como se puede ver la Figura 48), excepto en el Registro de Instrucciones. En este caso, se inyectaron 1000 fallos durante el primer ciclo de *cluster* y otros 1000 durante el segundo.
- **Duración de los fallos**. Transitorios, con una duración aleatoria seleccionada en el rango [½ periodo de SRU, 1 periodo de SRU].
- **Distribución de los fallos**. Uniforme.
- **Modelos de fallos**. *Bit-flip* (en elementos de memoria), *Pulse* (en lógica combinacional), Indeterminación y *Delay*.

Los resultados obtenidos se pueden ver en la Tabla 27.

	IR PCU (1st TDMA)	**IR PCU (2nd TDMA)**	**CRC**	**TCU**	**BG**
Errores activados	18.30 %	18.40 %	14.80 %	60.60 %	24.10 %
Errores no efectivos	22.40 %	20.65 %	72.30 %	51.16 %	29.25 %
Errores detectados	77.60 %	79.35 %	27.70 %	48.84 %	79.75 %
Errores recuperados	100 %	100 %	100 %	100 %	100 %
Errores no recuperados – Avería	0 %	0 %	0 %	0 %	0 %
Errores no detectados – Avería	0 %	0 %	0 %	0 %	0 %

Tabla 27. Resultados de la validación del controlador TTP/C-C2 [Blanc04].

Las principales conclusiones que se pueden obtener son:

- No se han encontrado averías, es decir, no existen violaciones del silencio ante fallos, tal y como se puede ver en los porcentajes marcados por *Errores no detectados – Avería* y *Errores no recuperados – Avería*. Además, todos los errores detectados han sido recuperados, como se ve en el porcentaje *Errores recuperados*.
- Los porcentajes de errores activados presentan una gran variabilidad, siendo la TCU el punto más sensible de los probados, ya que presenta el mayor porcentaje de errores activados. Esta gran variabilidad puede deberse a los diferentes números de inyecciones realizadas en cada experimento, así como a que en algunos casos, este número de inyecciones sea pequeño.
- Si observamos los porcentajes de errores detectados, podemos ver que también existe una gran variabilidad. En este caso el CRC presenta el porcentaje más bajo, siendo la *Redundancia Intrínseca* del sistema la que soluciona la mayor parte de los errores. En cambio, en los experimentos del Registro de Instrucciones y del Guardián del *Bus*, son los mecanismos de detección de errores los que detectan la mayor parte de éstos.

En definitiva, podemos concluir que el error del algoritmo del *clique avoidance* de la primera versión del controlador *TTP/C* ha sido corregido en la segunda versión.

7.4.1.3 Comparación y/o combinación de técnicas de inyección

A lo largo de los proyectos mencionados en el punto 7.2, uno de los resultados más comunes es la comparación y/o la combinación de diferentes técnicas de inyección de fallos, ya que se dispone del mismo sistema para realizar los diferentes experimentos de inyección. Por ejemplo, en [Karlsson94] se utiliza la inyección de fallos basada en simulación para estudiar más en detalle los fenómenos observados en experimentos de inyección de fallos mediante iones pesados. En [Folkesson98] se comparan la inyección de fallos basada en simulación con la inyección de fallos implementada mediante *Scan-Chain*, y en [Folkesson99] se hace una extensa comparación de diferentes técnicas de inyección de fallos física. Por último, en [Blanc01] se presenta un estudio donde se evalúa un sistema empotrado distribuido de control utilizando la inyección de fallos implementada por *software* y la inyección de fallos a nivel de *pin*, y en el que se muestra la complementariedad de ambas técnicas, y en [Blanc02a] se combinan tres técnicas de inyección de fallos (inyección de fallos a nivel de *pin*, inyección de fallos implementada por *software* e inyección de fallos mediante iones pesados) para evaluar el protocolo *TTP/C*.

En este punto se detallarán los experimentos realizados durante la comparación y/o combinación de técnicas de inyección de fallos realizadas durante el proyecto *FIT*. En este caso fueron la inyección de fallos basada en VHDL y la inyección de fallos a nivel de *pin*. Gracias a esta colaboración, se encontraron dos errores más. En primer lugar, se encontró otro error de diseño, concretamente en el algoritmo de actualización de la señal de vida del *TTP/C-C2* (en inglés, *Life-sign algorithm*) [Blanc04]. El otro error encontrado es un error arbitrario en el *TTP/C-C1*. Concretamente, es un error ligeramente fuera de las especificaciones (del inglés, *Slightly-Off Specification Error*) [Blanc02b].

Antes de continuar, hay que mencionar a **AFIT**, la herramienta de inyección de fallos a nivel de *pin* que ha sido desarrollada por el Grupo de Sistemas Tolerantes a Fallos de la Universidad Politécnica de Valencia y que se ha utilizado (y mejorado) durante el proyecto *FIT*. AFIT fue diseñada como una herramienta modular externa, con un computador personal que maneja la interfaz entre el usuario y el inyector. Los modelos de fallos que puede inyectar son inversiones de valor (del inglés *pulse*, tanto transitorios como intermitentes) y *stuck-at* permanentes, simples y múltiples [PGil92, PGil97, Martínez99, PGil03].

7.4.1.3.1 Error de diseño: algoritmo de actualización de la señal de vida

En este caso, se utilizaron AFIT y VFIT para inyectar fallos en el controlador *TTP/C-C2* [Blanc04]. Estos experimentos tienen las mismas hipótesis de fallos que los experimentos descritos en el punto 7.4.1.2.1:

- Un fallo simple no perturbará el funcionamiento del sistema completo.
- El nodo enviará mensajes correctos o no enviará nada (silencio ante fallos).

Existen dos tipos de datos intercambiados entre el controlador *TTP/C* y el *host*: mensajes e información de control y/o estado. Estos datos se intercambian a través de la CNI. Dentro del área de control/estado de la CNI, están los registros de señal de vida (en inglés, *life-sign registers*). Si el *host* está activo, debe actualizar periódicamente su registro de señal de vida con el fin de notificar al controlador *TTP/C* que los mensajes todavía están siendo escritos y/o leídos, es decir, que el *host* todavía está vivo.

En el caso de que el controlador *TTP/C* detecte un error en la actualización de la señal de vida del *host*, cambia del estado *activo* al *pasivo*, y espera a que el *host* esté preparado de nuevo para la ejecución de la aplicación. Además, el controlador activará la señal de interrupción del controlador *TTP/C*, que es la única señal externa entre el controlador y el *host*. La interrupción informa al *host* que el controlador ha detectado un error.

Durante los experimentos de inyección a nivel de *pin*, los fallos inyectados modificaban el valor almacenado en el registro de señal de vida mediante la inyección en los *buses* de datos y de direcciones. La Figura 64 y la Tabla 28 muestran los casos analizados.

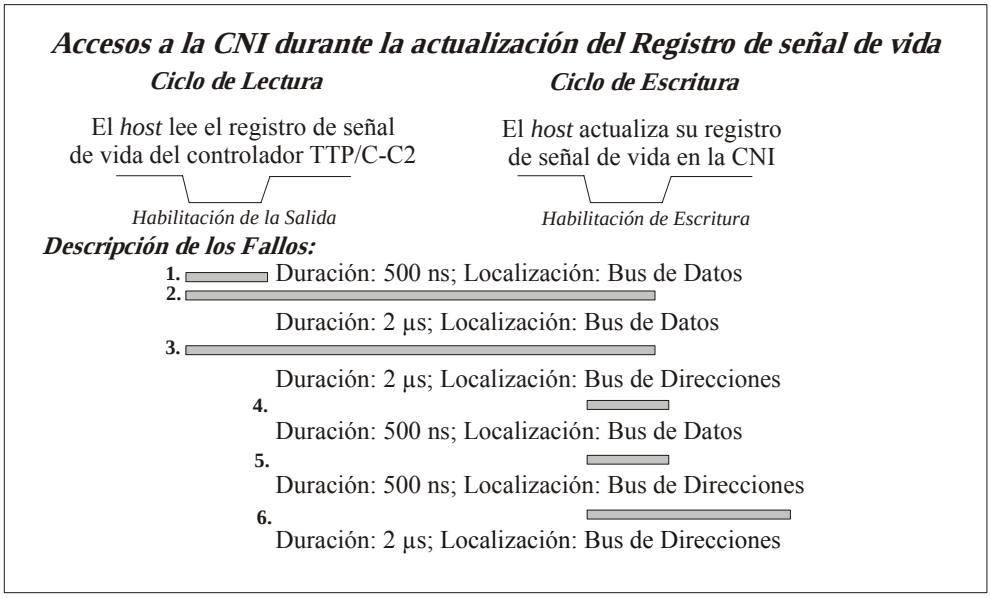


Figura 64. Inyecciones durante el algoritmo de actualización de la señal de vida en el controlador TTP/C-C2 [Blanc04].

Caso	Duración	Tipo fallo	Localización	Instante
1	500 ns	Fallo Transitorio Corto	*Bus* de datos	Ciclo de lectura
2	2 µs	Fallo Transitorio Largo	*Bus* de datos	Ciclo de lectura
3	2 µs	Fallo Transitorio Largo	*Bus* de direcciones	Ciclo de lectura
4	500 ns	Fallo Transitorio Corto	*Bus* de datos	Ciclo de escritura
5	500 ns	Fallo Transitorio Corto	*Bus* de direcciones	Ciclo de escritura
6	2 µs	Fallo Transitorio Largo	*Bus* de direcciones	Ciclo de escritura

Tabla 28. Inyecciones durante el algoritmo de actualización de la señal de vida en el controlador TTP/C-C2 [Blanc04].

El resultado de las distintas inyecciones varía según los casos. Los casos 2 y 3 provocan excepciones del *host*, ya que los fallos inyectados perturban los *buses* del *host*. En el caso 6 es muy común observar excepciones del sistema operativo. Los casos 1 y 4 provocan un *bit-flip* en los datos y en consecuencia, el *host* actualiza su registro de señal de vida con un valor erróneo. Por último, en el caso 5, el *host* no actualiza el registro.

Con el fin de realizar un análisis detallado de los casos 1, 4 y 5, se utilizó la inyección de fallos basada en VHDL para reproducir estos experimentos y comparar los resultados con la inyección de fallos a nivel de *pin*.

Una vez realizada la inyección, durante el control de las líneas de transmisión del nodo con el fallo inyectado, se observó que el nodo no transmitía ningún mensaje por ninguno de los dos canales durante la siguiente ronda de TDMA después de la inyección del fallo. Debido a este silencio del nodo con el fallo inyectado, el resto de miembros del *cluster* marcan a este nodo como *inactivo*. Esto significa que el controlador *TTP/C* ha detectado el error del *host* en la actualización del registro de señal de vida y no transmite nada, de acuerdo a la condición de silencio ante fallos.

Sin embargo, la línea de interrupción entre el controlador *TTP/C* y el *host* debería ser activada por el controlador, avisando al *host* de la detección del error. En cambio, lo que se observó fue que el bit de estado correspondiente del registro de interrupciones no se actualizaba, con lo que la línea de interrupción no se llega a activar, siendo este hecho una divergencia con

las especificaciones [TTP/C99]. Es decir, un fallo simple inyectado durante el acceso del controlador o del *host* al registro de señal de vida provoca un silencio en un periodo de SRU que no es detectado. La Figura 65-a muestra el funcionamiento normal del sistema, mientras que la Figura 65-b muestra el error.

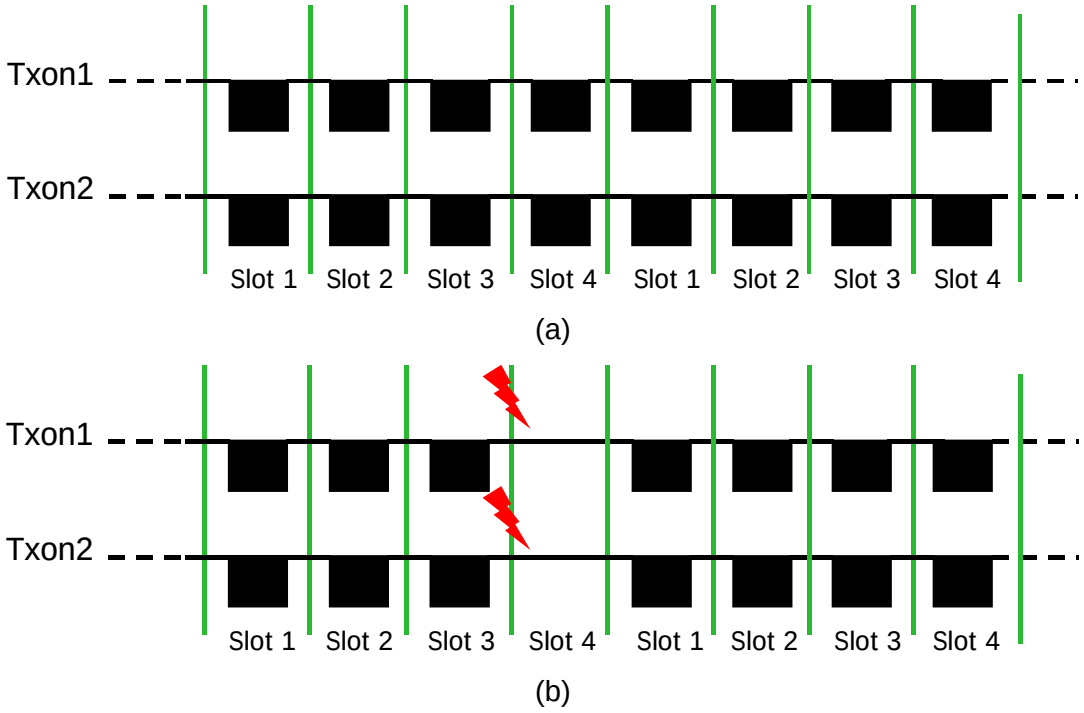


Figura 65. Error del algoritmo de actualización del registro de señal de vida.
(a) Funcionamiento normal. (b) Funcionamiento con fallo.

Básicamente, la secuencia de error es la siguiente:

- Se inyecta un fallo en el *Bus* de Datos sincronizado con el acceso al registro de señal de vida.
- El nodo con el fallo inyectado no transmite, con lo que se debería activar una interrupción.
- El resto de nodos del sistema marcan a este nodo como *inactivo*.
- Sin embargo, durante el siguiente periodo, el nodo con fallo transmite.
- En este punto, el resto de nodos del sistema marcan a este nodo como *activo*.
- Es decir, el fallo se ha propagado desde la CNI hasta el controlador de comunicaciones.

Como ya se ha comentado, este fallo también se ha comprobado con la inyección de fallos a nivel de *pin* sobre el prototipo real. Todo esto es considerado como un error en el diseño e implementación de un algoritmo específico, el cual podría haber sido fácilmente detectado en el modelo en VHDL mediante VFIT, antes de que el error se propagase al prototipo.

7.4.1.3.2 Error arbitrario: error ligeramente fuera de las especificaciones

Este error también se detectó durante la colaboración entre la inyección de fallos basada en VHDL y la inyección de fallos a nivel de *pin*. En este caso se hizo un estudio de las hipótesis de fallos del controlador *TTP/C* [Blanc02b]. Estas hipótesis de fallos para fallos físicos externos se define en las especificaciones del controlador *TTP/C* de la siguiente manera [TTP/C99]:

Si un fallo externo impacta solamente en un único subsistema TTP/C, el sistema TTP/C tolerará este fallo. Fallos múltiples durante un período de SRU que destruyan los dos mensajes en ambos canales serán tolerados en un cluster apropiadamente configurado si la distancia

temporal entre el final del período de SRU de la primera réplica y el comienzo del período de SRU de la segunda réplica es más larga que la duración del fallo transitorio.

Un *cluster* apropiadamente configurado requiere la replicación de nodos. Así, en términos de fallos múltiples, para un *cluster* no apropiadamente configurado se asume que los fallos transitorios son más cortos que un período de SRU. Debido a que estamos trabajando con una topología de *bus*, se asume que los fallos transitorios cubiertos por las hipótesis de fallos están contenidos dentro de los límites de un período de SRU. Es decir, la duración de los fallos inyectados debe ser menor que la duración de un periodo de SRU.

Al igual que en el punto anterior, y con el fin de poder comprobar el efecto de los fallos, éstos se inyectaron en los mismos lugares con las dos técnicas. En particular, se inyectaron fallos externos que corrompieran directamente los canales de transmisión. Se realizaron dos experimentos de inyección de fallos. En el primer experimento se inyectaron fallos transitorios simples en un único canal. En concreto, se inyectaron fallos en líneas externas accesibles mediante AFIT, es decir, se inyectaron fallos en las líneas de recepción y transmisión, en la línea de habilitación de la transmisión del Guardián del *Bus*, y en la línea de habilitación de la salida. Los modelos de fallos inyectados fueron el *stuck-at* (‘0’, ‘1’) utilizando AFIT, e indeterminación utilizando VFIT.

Como ya se ha comentado, el Guardián del *Bus* local es una unidad independiente que habilita al nodo para transmitir mensajes durante su periodo de transmisión. El *bus* consiste en dos canales replicados que forman dos unidades independientes de contención de fallos [Bauer01]. Un nodo tiene un módulo de transmisión y otro de recepción dedicados a cada canal, e independientes entre sí. Los resultados de los experimentos realizados con ambas técnicas muestran el mismo resultado: únicamente los mensajes enviados a través del canal con fallo se marcan como mensajes no válidos. Así pues, la llegada de un mensaje erróneo por uno de los canales no afecta al nivel de aplicación. Además, el nodo podrá ejecutar todos los servicios del protocolo gracias a los mensajes que llegan por el canal sin fallos.

Inyectando un *stuck-at* ‘0’ con AFIT en las líneas de control, se detiene la transmisión del mensaje durante la duración del fallo. Si la duración del fallo cubre toda la transmisión, no se transmite nada. El resto de nodos leen un mensaje vacío por el canal con fallos. Si la duración del fallo es más corta que la duración de la transmisión, se recibe un mensaje incorrecto, fácilmente detectable (por ejemplo, por el CRC). Un fallo del tipo *stuck-at* ‘1’ provoca errores si ocurre durante la recepción de un mensaje. Este caso es similar a un fallo en la línea de recepción. Con VFIT se observó que el error activaba el mecanismo de Error de Protocolo, perdiendo el nodo con fallo su pertenencia al *cluster*. Sin embargo, al tener los canales replicados e inyectar fallos simples, el mensaje corrompido es marcado como no válido por el nodo con fallos, mientras que el mensaje recibido por el canal sin fallos es utilizado por los servicios del protocolo, permitiendo que el nodo con fallos permanezca como miembro activo del *cluster*.

Respecto al segundo experimento, se inyectaron fallos múltiples que perturbaron los dos canales. En este caso, se identificaron cinco escenarios de averías [Kopetz01]:

1. Avería de enlace saliente (del inglés, *Outgoing link failure*).
2. Avería del *Babbling idiot*.
3. Avería de enmascaramiento (del inglés, *Masquerading failure*).
4. Avería ligeramente fuera de las especificaciones (del inglés, *Slightly-off-specification failure*).
5. Avería por proximidad espacial (del inglés, *Spatial proximity failure*).

El sistema debe tolerar los cinco escenarios de averías identificados siempre que se produzca un fallo que esté dentro de las hipótesis de fallos.

Como se ha comentado anteriormente, el *TTP/C* envía cada mensaje de forma replicada a través de los *buses* del sistema de comunicaciones. La Figura 66-a muestra una posible vista de ambos canales durante una ronda de TDMA. Como se puede ver, durante cada periodo de SRU se envían los mensajes replicados, utilizando para ello ambos canales. Dentro de un periodo de SRU se pueden identificar cinco áreas diferentes, tal y como se muestra en la Figura 66-b. Estas áreas son:

- Area 1. Desde el inicio del periodo hasta el inicio de la transmisión del mensaje.
- Area 2. Transmisión del mensaje.
- Area 3. Tiempo restante hasta que el Guardián del *Bus* local deshabilita la salida del controlador.
- Area 4. Tiempo restante del período de SRU asignado al nodo.
- Area 5. *Inter Frame Gap* (IFG) entre dos fases de transmisión consecutivas.

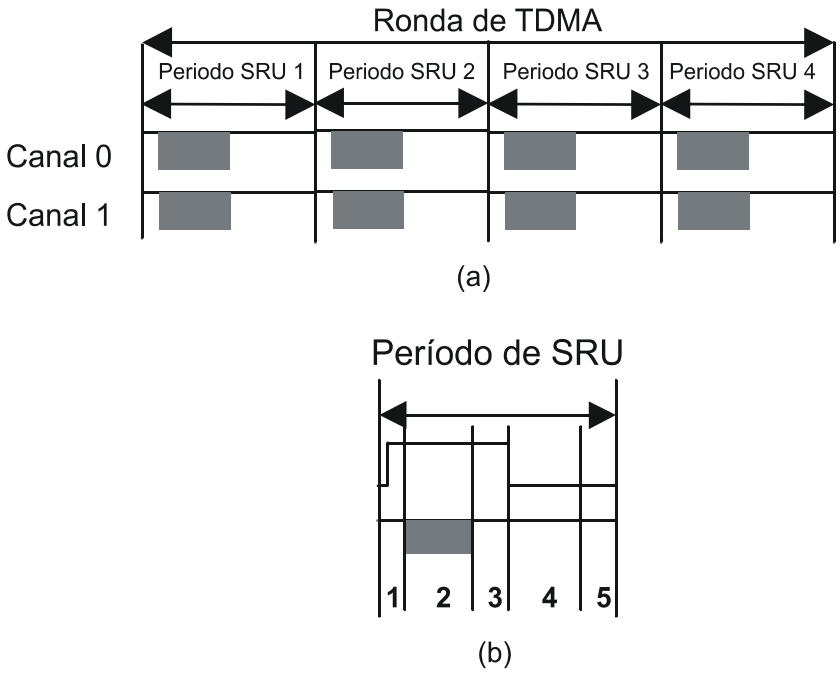


Figura 66. (a) Vista de los canales replicados durante una ronda de TDMA. (b) Areas diferentes en un periodo de SRU. [Blanc02b].

Así pues, dependiendo del área donde se inyecten los fallos, es posible llevar al sistema a cuatro de los escenarios de averías. El quinto escenario, la avería por proximidad espacial, depende del diseñador del sistema y de cómo distribuye los diferentes módulos y nodos en un avión, coche, etc. Los resultados obtenidos utilizando las dos técnicas de inyección presentaban ligeras diferencias, aunque seguían la misma tendencia general. A continuación se describirán los efectos de las inyecciones en las cinco áreas respecto a cuatro de los escenarios de averías.

7.4.1.3.2.1 Avería de enlace saliente (del inglés, Outgoing link failure)

Este tipo de avería ocurre cuando un nodo no puede enviar un mensaje al resto de nodos del *cluster*. Este escenario se puede generar con ambas herramientas mediante la inyección de un fallo de tipo *stuck-at* '0' en la línea del Guardián del *Bus* (ver Figura 47). El fallo inyectado debe durar lo mismo que el período de SRU, lo cual inhabilita la transmisión del nodo con el fallo inyectado durante su período de SRU, no transmitiendo ningún mensaje durante la ronda de TDMA actual. Como ya se ha comentado, las dos herramientas provocaron el mismo escenario. Los nodos activos reconocen este escenario de avería mediante el algoritmo de pertenencia.

7.4.1.3.2.2 Avería del babbling idiot

Este tipo de avería se produce cuando un nodo envía mensajes en instantes arbitrarios de tiempo, en los cuales no debería enviar nada. En este caso, los fallos inyectados que perturban el *bus* no pueden crear un mensaje correcto, por lo que la transmisión de un mensaje correcto en un instante de tiempo incorrecto se contempla en las averías de enmascaramiento.

7.4.1.3.2.3 Avería de enmascaramiento (del inglés, masquerading failure)

El nodo asume una identidad incorrecta y ocupa el periodo de SRU asignado a otro nodo. En este escenario, el nodo con el fallo inyectado envía mensajes correctamente formateados. Este fallo no se pudo producir, ya que la inyección en una línea de salida, como por ejemplo una línea de transmisión, no puede provocar el cambio de identidad del nodo. En este caso se inyectaron fallos en la interfaz entre la MEDL y el controlador *TTP/C-C1*. En cualquier caso, no se observó el escenario de la avería de enmascaramiento.

7.4.1.3.2.4 Avería ligeramente fuera de las especificaciones (del inglés, Slightly-off-specification failure)

Una avería ligeramente fuera de las especificaciones es un escenario asimétrico donde el efecto del fallo es interpretado de forma diferente por los nodos del sistema. En el caso de la inyección de fallos basada en VHDL, se puede utilizar el modelo de fallo *indeterminación* para reproducir este escenario. En este caso se utilizaron tres configuraciones en un *cluster*: cuatro, cinco y seis nodos, obteniéndose los mismos resultados con las tres configuraciones.

En un *cluster* con seis nodos, se inyectaron fallos transitorios con una duración de 2 μs en la línea de recepción (como se acaba de comentar, se inyectaron *indeterminaciones*). Los fallos se inyectaron en dos nodos seleccionados aleatoriamente, excluyendo al nodo propietario del periodo de SRU actual, el cual no lee el *bus*. Este experimento se repitió inyectando fallos en tres y cuatro nodos.

Únicamente los nodos inyectados provocaron que el nodo propietario del periodo de SRU actual perdiera su pertenencia al *cluster*. Es decir, los nodos inyectados marcan en su vector de pertenencia que el nodo propietario del periodo de SRU actual no pertenece al *cluster*. Con esta acción, se divide el *cluster* en dos grupos con visiones diferentes del sistema, representadas en los vectores de pertenencia de los distintos nodos. Por ejemplo, los dos nodos inyectados marcan al emisor como no válido, mientras que los tres nodos restantes marcan al emisor como válido. Esta inconsistencia es resuelta por el algoritmo de *clique avoidance* en un TDMA, mientras que el algoritmo de pertenencia decide la pertenencia del emisor. El grupo mayor sobrevive mientras que el resto cumple la condición de silencio ante fallos mediante la identificación de un error de reconocimiento y cambiando al estado de *congelado*. Todo esto es calculado en dos TDMA (el TDMA en el que se da la avería ligeramente fuera de las especificaciones y el siguiente). Si existe un empate, es decir, el mismo número de nodos en los grupos que marcan si el emisor es o no válido, el *cluster* entero identifica un error de reconocimiento, excepto el emisor, el cual señala un apagón del sistema de comunicaciones.

Además, gracias a la precisión que la técnica de inyección de fallos basada en VHDL puede conseguir, las inyecciones en el área 1 se pudieron dividir en tres partes, tal y como se puede ver en la Figura 67. Se inyectaron fallos transitorios con una duración de 2 μs. En este caso, esta duración es igual a la duración de 2 *macroticks*[54]. Se ha utilizado una configuración con cuatro nodos (la configuración mínima para la correcta sincronización del reloj).

[54] El *macrotick* compone la granularidad del tiempo global.

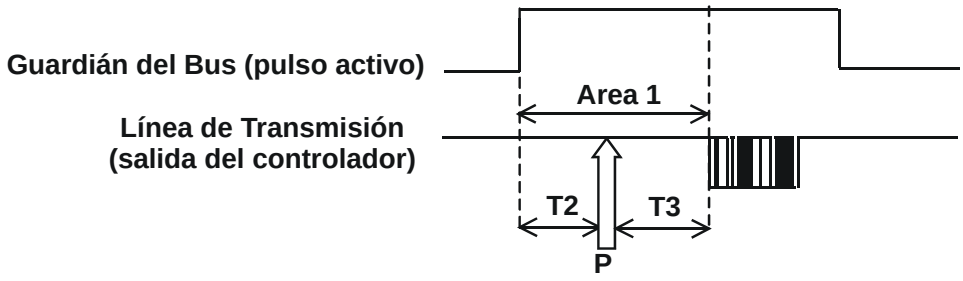


Figura 67. División del área 1 [Blanc02b].

Los efectos de la inyección en el área 1 fueron los siguientes:

- Los fallos inyectados en T2 no tienen efecto en el modelo.
- Los fallos inyectados en T3 se toman como parte del mensaje a enviar, con lo que el resto de nodos del *cluster* marcarán este mensaje como no válido, causando que el nodo inyectado pierda su marca de pertenencia, es decir, deja de pertenecer al *cluster*.
- Los fallos inyectados en P se comportan de forma diferente. Durante la duración del área 1 (T2 + T3 + P), todos los nodos del *cluster* esperan un nuevo mensaje. El fallo transitorio inyectado en P es tomado como el nuevo mensaje esperado, pero al no tener el formato correcto, es tomado como un mensaje no válido. En consecuencia, se incrementa el contador de mensajes no válidos. Sin embargo, hay tiempo suficiente entre el fallo y el mensaje correcto (periodo T3 de la Figura 67) para tomar la transmisión del mensaje real como una transmisión no esperada de otro mensaje. Esta segunda transmisión incrementa otra vez el contador de mensajes no válidos. Con esta actualización, el contador de mensajes no válidos alcanza el mismo valor que el contador de mensajes válidos, y debido al algoritmo del *clique avoidance*, los tres receptores restantes activan el error de reconocimiento. El emisor, siendo el único nodo activo en el *cluster* activa el error de apagón del sistema de comunicaciones, con lo que los cuatro nodos del sistema paran la comunicación y cambian a un estado de *congelado*.

  Si se utiliza un *cluster* con seis nodos (un número mayor que la configuración mínima) el contador de mensajes correctos es mayor que el contador de mensajes no válidos, cambiando el efecto del fallo por una pérdida de pertenencia del nodo emisor.

  Si los tiempos locales de los nodos del *cluster* fueran diferentes entre sí, este error puede causar un fallo asimétrico.

Durante las inyecciones en el área 2 (ver Figura 66-b), los receptores marcaron el mensaje recibido como no válido, causando la pérdida de pertenencia del emisor. En el área 3 y en la 5, los fallos no tuvieron efecto en el sistema de comunicaciones.

Por último, aunque durante el área 4, el emisor ya ha terminado la transmisión del mensaje, el *bus* todavía lo tiene asignado el emisor. Un fallo transitorio es interpretado como una segunda transmisión de mensaje con un formato incorrecto, por lo que los receptores lo marcan como no válido provocando que el emisor pierda su pertenencia al *cluster*.

En resumen, se puede decir que un fallo que afecta a ambas líneas de transmisión causa un apagón en la línea de comunicaciones. Este problema se da cuando se inyectan los fallos en un determinado instante de tiempo (como se puede ver en la Figura 67). En ese instante, el fallo es tomado como un nuevo mensaje con un formato incorrecto, provocando el error.

Una posible solución a esta avería sería la definición de un periodo máximo de tiempo entre la ventana de comienzo de transmisión hasta el instante de envío del mensaje. Este periodo de tiempo es dependiente del diseño y estaría definido por el retardo que presenta la detección por parte del controlador de un nuevo mensaje recibido hasta que este mensaje es marcado como no válido debido a un prolongado e inesperado silencio en el *bus* [FIT02b].

7.5 Resumen. Conclusiones y líneas abiertas de investigación

En este capítulo se ha presentado un breve estudio del *estado del arte* de la validación de sistemas distribuidos de tiempo real tolerantes a fallos, para después describir el proyecto *FIT*. Durante este proyecto, financiado por la unión europea, se ha validado el controlador de comunicaciones *TTP/C*.

Una vez descrito el proyecto *FIT*, se ha pasado a detallar los diversos experimentos de inyección de fallos realizados sobre el modelo en VHDL de dos versiones del controlador *TTP/C*: el *C1* y el *C2*. El uso del VHDL durante la fase de diseño de un sistema permite una temprana validación del modelo, con el fin de detectar y corregir posibles fallos antes de implementar el prototipo real, gracias a la inyección de fallos basada en VHDL, ahorrando así tiempo y dinero.

Todos los experimentos de inyección de fallos se realizaron en función de los objetivos descritos por el proyecto *FIT*, que básicamente eran la determinación de la cobertura de detección de errores de los diferentes mecanismos de detección de errores del controlador *TTP/C*, la localización de debilidades en la arquitectura y sus posibles soluciones y la comparación de las diferentes técnicas de inyección de fallos.

En primer lugar, se llevaron a cabo varios experimentos de inyección de fallos con el fin de realizar una validación general de ambos controladores, comprobándose la eficiencia de los mecanismos de detección de errores. Durante esta validación general, se comprobó que la mayor parte de los errores eran detectados por los mecanismos autocomprobantes internos y por el mecanismo error de protocolo, tal y como se preveía.

Otro resultado que se obtuvo durante estos experimentos fue la detección de una implementación errónea del algoritmo del *clique avoidance* en el controlador *TTP/C-C1*. Esta mala implementación provocaba un apagón del sistema de comunicaciones. Es decir, se producía una violación del silencio ante fallos, ya que un nodo que producía un mensaje erróneo provocaba que el resto de nodos dejasen de enviar mensajes, lo que quiere decir que un defecto del microcódigo (o *firmware* en inglés) provocaba una avería grave del sistema.

Otro punto a resaltar del proyecto *FIT* fue la comparación de las distintas técnicas de inyección de fallos, así como la cooperación que se podían realizar entre ellas. Una prueba fue la combinación de la inyección de fallos basada en VHDL y la inyección de fallos a nivel de *pin*. Esta combinación permitió una validación más completa y profunda del sistema, pudiéndose trazar los fallos inyectados con una técnica mediante la otra.

Con esta combinación se encontraron dos fallos más. En primer lugar, se pudo encontrar el fallo en el algoritmo de actualización de la señal de vida del controlador *TTP/C-C2*, donde un conjunto de fallos inyectados a nivel de *pin* revelaron una disparidad del prototipo respecto a las especificaciones. Este error se encontró al realizar un estudio de diferentes escenarios de fallos específicos. Repitiendo los mismos experimentos con la inyección de fallos basada en VHDL, se reprodujeron los mismos resultados.

El último error encontrado se ha definido como un *error ligeramente fuera de las especificaciones*. En un principio, se encontró un punto en el cual, al inyectar un fallo, se producía un apagón en el sistema de comunicaciones. En este caso, se llevó a cabo un estudio de la hipótesis de fallos del protocolo de comunicaciones del controlador *TTP/C-C1*. Las inyecciones de fallos incluyeron fallos externos simples y múltiples, específicamente localizados en las líneas de transmisión y control.

Existen aún varias líneas abiertas de investigación. Como se ha comentado, la aplicación que ejecutaba el sistema bajo prueba era una aplicación que simplemente activaba los diferentes módulos del sistema. La pregunta que surge es cómo afectaría una carga de trabajo real a los resultados de la validación. Otra línea que queda es la influencia de los modelos de fallos y de

las distintas técnicas de inyección. Sin embargo, el aplicar nuevas técnicas de inyección de fallos, como pueden ser los perturbadores o los mutantes, acarrea el buscar nuevos métodos que puedan acelerar las simulaciones. Este último punto es el gran problema de las técnicas de inyección basadas en simulación.

8 Conclusiones. Trabajo futuro

En este capítulo se realiza una recapitulación del trabajo expuesto en la presente tesis. En primer lugar, se extraerán las conclusiones más importantes del trabajo desarrollado, para después exponer las publicaciones a que, directa o indirectamente, ha dado lugar. Por último se abordarán las futuras líneas de investigación por las que se puede continuar el trabajo recientemente presentado.

8.1 Conclusiones

El trabajo de esta tesis se ha centrado en diferentes aspectos de la técnica de inyección de fallos mediante simulación de modelos en VHDL, incidiendo en las características generales de la técnica, y en las particulares de las diferentes variantes de implementación. Con el fin de probar las distintas técnicas de inyección de fallos en VHDL, éstas se han aplicado sobre diferentes sistemas modelados en este lenguaje, obteniéndose resultados y mejoras tanto de las técnicas como de los diferentes modelos bajo prueba.

8.1.1 Inyección de fallos sobre modelos en VHDL

En primer lugar se ha realizado un profundo estudio de la inyección de fallos como un método general para validar la Confiabilidad de los sistemas tolerantes a fallos (mediante el cálculo de parámetros como los coeficientes de cobertura en la detección y recuperación de errores, o los tiempos de latencia hasta la detección y recuperación de los errores). Las diferentes técnicas de inyección de fallos se pueden clasificar en tres: inyección física (o HWIFI), inyección implementada por *software* (o SWIFI) e inyección mediante simulación. Se ha realizado un profundo estudio del *estado del arte* de todas las técnicas de inyección, describiendo su método de trabajo, sus ventajas e inconvenientes, así como las herramientas y propuestas más importantes. También se ha estudiado la nueva tendencia que se está observando en estos últimos años, a la que se ha denominado *inyección de fallos híbrida*. Esta nueva corriente intenta combinar las ventajas de diferentes técnicas de inyección en una única herramienta, con la idea de permitir que sobre un mismo sistema se puedan aplicar varias técnicas de inyección que se complementen entre sí y poder obtener unos resultados mucho más fiables, evitando de esta manera las desventajas que surgen al aplicar una única técnica de inyección.

Una vez estudiadas las diferentes técnicas de inyección, se ha realizado un exhaustivo análisis de la inyección de fallos basada en VHDL. Las ventajas de esta técnica con respecto a las diferentes técnicas de inyección de fallos se pueden resumir en:

- La amplia utilización de este lenguaje en el diseño digital actual.
- Permite describir un sistema en distintos niveles de abstracción (puerta, RT, chip, algorítmico, sistema, etc.).
- Permite descripciones estructurales y comportamentales en un único elemento sintáctico.
- Algunos elementos de su semántica facilitan la inyección de fallos.

Sin embargo, la principal ventaja de las técnicas basadas en simulación radica en el hecho de que se pueden aplicar en la fase de diseño, con el consiguiente ahorro en costes de rediseño o reimplementación. Ofrece además altos niveles de genericidad, accesibilidad, controlabilidad y capacidad de automatización. No obstante, su mayor inconveniente es el elevado coste temporal de la realización de la inyección en sistemas y cargas complejos.

La presente tesis parte de algunos trabajos previos sobre simulación en VHDL realizados en el seno del GSTF, los cuales se pueden estudiar en las tesis de D. Daniel A. Gil Tomás y de D. J. Carlos Baraza Calvo. En el primer caso se establecen los fundamentos de la inyección de

fallos basada en VHDL, mientras que en el segundo se desarrolla una herramienta para la inyección de fallos en modelos VHDL de forma automática y se estudia la representatividad de los fallos a inyectar.

A partir del trabajo previo desarrollado, en la presente tesis se ha estudiado, implementado y mejorado la aplicación de los diferentes métodos de inyección de fallos en VHDL. Las técnicas estudiadas han sido la basada en las órdenes del simulador (perteneciente al grupo de técnicas en las no se modifica el modelo) y los perturbadores y los mutantes (ambas incluidas entre las técnicas que requieren modificar el código del modelo). Las tres técnicas elegidas tienen diferentes características en cuanto a facilidad de implementación, de automatización, modelos de fallos, etc. Como principales características de estas técnicas, podemos decir:

- La técnica de las órdenes del simulador se basa en la utilización de ciertas órdenes del simulador para controlar la simulación del modelo, alterando el valor y/o la temporización de los elementos internos del mismo. Es, con diferencia, la más sencilla de implementar y automatizar, siendo la técnica que se ejecuta de una forma más rápida. Además, el tiempo necesario para la inyección del fallo (parar la simulación, modificar la señal o variable, simular la duración del fallo y en el caso de las señales volver a parar la simulación y liberar la señal) es despreciable respecto al tiempo total de simulación. Sin embargo, tiene como principal inconveniente que el conjunto de modelos de fallos que se pueden aplicar es dependiente de las capacidades del simulador, y además este conjunto es el más reducido de las tres técnicas.
- Un perturbador se puede definir como un componente que se añade al modelo VHDL con la función de alterar el valor o las características temporales de una o más señales a la hora de inyectar un fallo. Esta técnica se aplica a modelos con descripciones estructurales. Una vez añadidos los perturbadores, éstos se controlan mediante entradas especiales que determinan si se inyecta o no un fallo, así como el modelo de fallo a inyectar. La principal ventaja de esta técnica, con respecto a las órdenes del simulador, es la ampliación de los modelos de fallos que se pueden inyectar. Sin embargo, esta técnica presenta cierta complejidad de implementación y automatización, especialmente en lo que se refiere a la elaboración de los componentes perturbadores y su inclusión en el modelo. Además, al añadir un cierto número de señales de control, el tamaño del fichero de traza aumenta. Estas señales se utilizan para la activación de uno de los perturbadores y para indicar el tipo de fallo (o perturbación) que se va a inyectar. En cuanto al tiempo de simulación, éste es mayor que el tiempo empleado con la técnica de las órdenes del simulador, debido principalmente a la simulación del modelo más la simulación de los perturbadores añadidos al mismo.
- Se puede definir a los mutantes como nuevos componentes VHDL que reemplazan a los componentes originales. Durante el funcionamiento normal, el componente mutado se comporta como el componente al que reemplaza. Sin embargo, cuando se inyecta el fallo, el componente mutado simula el comportamiento del componente original, pero en presencia de fallos. Esta técnica es adecuada principalmente para modelos con descripciones comportamentales (o estructurales con descripciones comportamentales en los niveles más bajos de la jerarquía). La inyección de un fallo consiste en alternar la simulación de la versión original del sistema con la correspondiente versión mutada. La implementación de esta técnica presenta problemas de sincronización entre la simulación del código sin y con fallo. También se produce un consumo desmesurado de tiempo, sobretodo a la hora de inyectar fallos transitorios. La ventaja principal de esta técnica es una gran variedad de modelos de fallos, muy superior a los de las otras dos.

Hay que destacar las aportaciones realizadas por esta tesis en la implementación de estas técnicas. Respecto a las órdenes del simulador, se han ampliado los modelos de fallos a inyectar, pasándose de los básicos *bit-flip* para fallos transitorios y *stuck-at* para fallos permanentes a un conjunto más complejo y elaborado de modelos de fallos, tanto transitorios como permanentes.

En cuanto a los perturbadores, a partir de los modelos básicos presentados en la literatura, se han desarrollado nuevos tipos de perturbadores, basándonos en dos ideas fundamentales. La primera era el conseguir una máxima extensión de la inyección de fallos en el sistema bajo prueba, destacando la ampliación realizada en el diseño de algunos de los perturbadores con el fin de poder inyectar fallos en *buses* y en ambas direcciones (es decir, tanto en buses unidireccionales como bidireccionales). La segunda idea consistió en la ampliación de los modelos de fallos utilizados hasta este momento en la literatura, lo que permitió superar los clásicos modelos de fallos comentados anteriormente para inyectar un nuevo conjunto de modelos de fallos más completo y elaborado.

Respecto a los mutantes, el logro más importante ha sido la posibilidad de inyectar fallos transitorios mediante esta técnica, resolviendo para ello una serie de problemas que provocaban que este tipo de fallos no se hubiesen podido inyectar antes. Al igual que con las técnicas anteriores, los nuevos modelos de fallos implementados e inyectados con esta técnica amplían las posibilidades de la misma.

8.1.2 Aplicación de la inyección de fallos basada en VHDL

Una vez que se han estudiado las tres técnicas de inyección de fallos basadas en VHDL, éstas se han aplicado sobre un microprocesador tolerante a fallos, realizando una serie de experimentos de inyección de fallos con el fin de calcular diferentes parámetros de la Confiabilidad, así como para estudiar la respuesta del sistema en presencia de diferentes modelos de fallos, tanto transitorios como permanentes. Si bien este primer sistema bajo prueba es un modelo con una finalidad docente, los diferentes experimentos han servido para probar las diferentes técnicas de inyección de fallos así como la validez de los nuevos modelos de fallos empleados. Las principales conclusiones que se han obtenido de estos experimentos han sido:

- Se han verificado las buenas prestaciones de las tres técnicas de inyección, así como la alta controlabilidad y observabilidad general de la inyección en VHDL. El uso combinado de las diferentes técnicas de inyección proporciona un método muy poderoso para validar sistemas modelados en diferentes niveles de abstracción.
- Para fallos transitorios, los valores de las coberturas de detección y recuperación muestran pocas diferencias con cualquiera de las tres técnicas. Se puede deducir que las coberturas para este tipo de fallos se pueden obtener con bastante exactitud con cualquiera de las técnicas.
- Respecto a la relación entre el tipo de modelo y las técnicas de inyección, se pueden realizar varias consideraciones. En las partes estructurales del modelo es preferible utilizar las técnicas de las órdenes del simulador o los perturbadores. A pesar de que los perturbadores son más difíciles de implementar, presentan una mayor capacidad para modelar fallos. En los componentes comportamentales se pueden utilizar las técnicas de los mutantes o las órdenes del simulador. En este caso, son los mutantes los que presentan una mayor capacidad para modelar fallos, ya que pueden usar toda la capacidad semántica y sintáctica del VHDL.
- Se han presentado nuevos modelos de fallos que mejoran los utilizados hasta ahora. Estos nuevos modelos de fallos utilizados tienen una incidencia mayor en el normal funcionamiento del sistema respecto a modelos de fallos utilizados clásicamente en la literatura. Los nuevos modelos de fallos son posibles gracias a las capacidades de programación que presenta el simulador de VHDL utilizado y a las capacidades semánticas del VHDL.
- La información presentada en los distintos experimentos de inyección puede ser utilizada para mejorar el diseño de los mecanismos de detección y recuperación del sistema bajo prueba, así como para optimizar los valores de las coberturas y las latencias.

Para poder realizar los distintos experimentos de inyección, se ha utilizado VFIT, una herramienta de inyección de fallos desarrollada por el GSTF. Durante el desarrollo de la presente tesis, se implementó el módulo analizador de VFIT, mejorando sus prestaciones a medida que los modelos VHDL utilizados durante los diferentes experimentos de inyección de fallos se volvían más complejos, ampliándose de esta manera el conjunto de los parámetros que se podían obtener.

8.1.3 Validación de la arquitectura Time-Triggered

Actualmente, y debido a la introducción de sistemas empotrados en aplicaciones críticas, los requerimientos para conseguir una alta Confiabilidad en este tipo de sistemas han crecido. Uno de los servicios esenciales proporcionados por este tipo de arquitecturas es la transmisión de información desde uno de los componentes del sistema distribuido a otro componente, por lo que para realizar esta comunicación es habitual tener un *bus*. De esta forma, el *bus* de comunicaciones se convierte en uno de los componentes principales del sistema, y el protocolo de comunicaciones utilizado en una parte esencial del mismo.

El siguiente paso en el desarrollo de la presente tesis ha sido el estudio de diferentes arquitecturas de *bus* para sistemas distribuidos de tiempo real tolerantes a fallos, incidiendo de manera especial en la arquitectura *Time-Triggered*. El protocolo *Time-Triggered* es un protocolo de comunicaciones integrado, diseñado para arquitecturas *Time-Triggered*. Este protocolo proporciona los diferentes servicios requeridos para la implementación de sistemas en tiempo real tolerantes a fallos: transmisión de mensajes predecible, reconocimiento implícito de mensajes para comunicaciones en grupo, sincronización de reloj, servicio de pertenencia, cambios rápidos de modo y control de redundancia. La implementación de estos servicios se realiza sin mensajes extra y con una sobrecarga mínima en el tamaño del mensaje.

Se ha prestado una atención particular a la validación de sistemas empotrados con funciones críticas mediante la inyección de fallos, realizando un estudio de los diferentes proyectos que ha patrocinado la Unión Europea. Es en el marco de uno de estos proyectos donde se han llevado a cabo el segundo grupo de experimentos dentro de la presente tesis, en el que se han inyectado fallos transitorios en dos modelos en VHDL del controlador *TTP/C*. Este controlador está basado en la arquitectura *Time-Triggered*. El proyecto en cuestión se denominó *FIT* (*Fault Injection for TTA*, IST-1999-10748). Básicamente, los objetivos del proyecto *FIT* se pueden resumir en tres:

- Determinación de la cobertura de detección de errores de los diferentes mecanismos de detección de errores del controlador *TTP/C*.
- Localización de debilidades en la arquitectura y sus posibles soluciones.
- Comparación de las diferentes técnicas de inyección de fallos.

Con el fin de determinar la cobertura y la eficiencia de los diferentes mecanismos de detección de errores implementados, se llevaron a cabo varios experimentos de inyección de fallos cuyo objetivo principal era realizar una validación general de dos controladores basados en la arquitectura *Time-Triggered*, y descritos en VHDL. Durante esta validación general, se comprobó que la mayor parte de los errores eran detectados por los mecanismos autocomprobantes internos y por el mecanismo error de protocolo, tal y como se preveía.

Estos experimentos de inyección de fallos también permitieron localizar diferentes debilidades en el modelo VHDL. En particular, durante esta primera serie de experimentos se detectó una implementación errónea del algoritmo del *clique avoidance* en la versión del controlador *TTP/C-C1*. Este error provocaba una caída del sistema de comunicaciones. Es decir, se producía una violación del silencio ante fallos ya que un nodo que producía un mensaje erróneo provocaba que el resto de nodos dejasen de enviar mensajes. Gracias a la inyección de fallos mediante VHDL, se pudo detectar este problema y se pudo proponer la solución del mismo.

Otro punto a resaltar del proyecto *FIT* fue la posibilidad de realizar la comparación de las distintas técnicas de inyección de fallos utilizadas durante dicho proyecto, así como la cooperación que se podían realizar entre ellas. La comparación de técnicas ha permitido establecer una serie de pautas a la hora de utilizar una u otra técnica. Sin embargo, los resultados más espectaculares se han conseguido mediante la combinación de técnicas. En esta tesis, se han realizado una serie de experimentos utilizando la inyección de fallos basada en VHDL y la inyección de fallos a nivel de *pin*. Esta combinación permitió una validación más completa y profunda del sistema, pudiéndose trazar los fallos inyectados con una técnica mediante la otra.

Gracias a esta colaboración, se identificaron más debilidades en la arquitectura bajo prueba. En particular, se pudo encontrar un error en el *algoritmo de actualización de la señal de vida* del controlador *TTP/C-C2* y un *error ligeramente fuera de las especificaciones* en el controlador *TTP/C-C1*. El error en el *algoritmo de actualización de la señal de vida* del controlador *TTP/C-C2* se detectó al inyectar un conjunto de fallos mediante la técnica de inyección de fallos a nivel de *pin*. Este error se encontró al realizar un estudio de diferentes escenarios de fallos específicos. La inyección de fallos a nivel de *pin* revelaron una disparidad del prototipo respecto a las especificaciones. Gracias a la gran controlabilidad y observabilidad que permite la inyección de fallos en VHDL, al repetir los mismos experimentos con VFIT se pudieron trazar e investigar con profundidad este error, confirmándose los mismos resultados.

El último fallo de diseño encontrado se ha definido como un *error ligeramente fuera de las especificaciones*. En este caso, se encontró un punto en el cual, al inyectar un fallo con ambas técnicas, se producía una caída del sistema de comunicaciones. Para investigar este error, se realizó un estudio de la hipótesis de fallos del protocolo de comunicaciones del controlador *TTP/C-C1*. Las inyecciones de fallos incluyeron fallos externos simples y múltiples, específicamente localizados en las líneas de transmisión y control.

8.2 Publicaciones

Como resultado del trabajo desarrollado para la realización de la presente tesis, se han conseguido un número considerable de publicaciones, los cuales incluyen un capítulo de un libre y varios artículos en revistas y ponencias en congresos, todos ellos de relevancia en el campo de la inyección de fallos en VHDL. También es de destacar el hecho de que algunas de las publicaciones en las que ha participado el autor de esta tesis han sido referenciados en trabajos publicados por prestigiosos investigadores en el campo. A continuación se referencian las publicaciones realizadas.

8.2.1 Capítulo Libro

"VHDL Simulation-Based Fault Injection Techniques"; D. Gil, J.C. Baraza, J. Gracia, P.J. Gil; Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation, Edited by Alfredo Benso and Paolo Prinetto, Kluwer Academic Publishers, ISBN: 1-4020-7589-8, pp. 159-176, Octubre 2003.

8.2.2 Revistas

"A Prototype of a VHDL-Based Fault Injection Tool. Description and Application"; J.C. Baraza, J. Gracia, D. Gil, P.J. Gil; Journal of Systems Architecture, I.S.S.N. 1383-7621, 47(10):847-867, Abril 2002.

"Study, Comparison and Application of different VHDL-Based Fault Injection Techniques for the Experimental Validation of a Fault-Tolerant System"; D. Gil, J. Gracia, J.C. Baraza, P.J. Gil, Microelectronics Journal, Special Section on Defect and Fault Tolerance in VLSI Systems, ISBN: 0026-2692, Vol 34(1):41-51, Enero 2003.

"Using VHDL-Based Fault Injection for the Early Diagnosis of a *TTP/C* Controller"; J. Gracia, J.C. Baraza, D. Gil, P. Gil; Transactions on Information and Systems, Special Issue on Dependable Computing, The Institute of Electronics, Information and Communication Engineers, ISSN: 0916-8532, Vol. E86-D, Nº 12, pp. 2634-2641, Diciembre 2003.

8.2.3 Congresos

"Application Of Different VHDL-Based Fault Injection Techniques to the Validation of a Fault-Tolerant Microcomputer System"; J. Gracia, D. Gil, J. C. Baraza, P. J. Gil; FastAbstracts of the International Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8 – DSN'2000), pp. B-54 - B-55, New York, NY, USA, Junio 2000.

"A Study of the effects of Transient Fault Injection into the VHDL Model of a Fault-Tolerant Microcomputer System"; D. Gil, J. Gracia, J.C. Baraza, P. J. Gil; Procs. 6th IEEE International On-Line Testing Workshop (IOLTW'2000), ISBN: 0-7695-0646-1, pp. 73-79, Palma de Mallorca, España, Julio 2000.

"A Prototype of a VHDL-Based Fault Injection Tool"; J. C. Baraza, J. Gracia, D. Gil, P. J. Gil; in IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2000), ISBN: 0-7695-0719-0, pp. 396-404, Yamanashi, Japan, Octubre 2000.

"A Study of the Experimental Validation of Fault Tolerant Systems Using Different VHDL-Based Fault Injection Techniques"; J. Gracia, J.C Baraza, D. Gil and P.J. Gil; Procs. of the 7th IEEE On-Line Testing Workshop (IOLTW'2001), ISBN: 0-7695-1290-9, pp. 140, Taormina, Italia, Julio 2001.

"Comparison and Application of different VHDL-Based Fault Injection Techniques"; J. Gracia, J.C. Baraza, D. Gil, P.J. Gil; Procs. 2001 IEEE Int. Symposium on Defect and Fault

Tolerance in VLSI Systems (DFT 2001), ISBN: 0-7695-1203-8, pp. 233-241, San Francisco (USA), Octubre 2001.

"VFIT: Una herramienta automática para la inyección de fallos en VHDL"; J. Gracia, S. Blanc, J.C. Baraza, D. Gil, P.J. Gil; Actas Seminario Anual de Automática, Electrónica Industrial e Instrumentación (SAAEI'02), ISBN: 84-813-8514-X, pp. II-289–II-292, Alcalá de Henares (Madrid), Septiembre 2002.

"Studying Hardware Fault Representativeness with VHDL Models"; J. Gracia, D. Gil, L. Lemus, P. J. Gil, XVII Conference on Design of Circuits and Integrated Systems (DCIS 2002), ISBN: 84-8102-311-6, pp. 33-38, Santander, Noviembre 2002.

"A Fault Hypothesis Study on the *TTP/C* using VHDL-based and Pin-level Fault Injection Techniques"; S. Blanc, J. Gracia, P.J. Gil; Procs. IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2002), ISBN: 0-7695-1831-1, pp. 254-262, Vancouver, Canadá, Noviembre 2002.

"Using VHDL-Based Fault Injection to exercise Error Detection Mechanisms in the Time-Triggered Architecture"; J. Gracia, D. Gil, J.C. Baraza, P.J. Gil; Procs. 2002 Pacific Rim International Symposium on Dependable Computing (PRDC 2002), ISBN: 0-7695-1852-4, pp. 316-320, Tsukuba (Japón), Diciembre 2002.

"Early Diagnosis of Hard Real-Time Fault-Tolerant Embedded Systems"; J. Gracia, J.C. Baraza, D. Gil, P.J. Gil; Procs. 6th International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2003), ISBN: 83-7143-557-6, pp.157-164, Poznán (Polonia), Abril 2003.

"Experiences during the Experimental Validation of the Time-Triggered Architecture"; S. Blanc, J. Gracia, P.J. Gil; Procs. Design, Automation and Test in Europe (DATE'04), ISBN: 0-7695-2085-5-3, pp. 30256-30261, CNIT la Défense, París, Febrero 2004.

8.2.4 Publicaciones con referencia a nuestros trabajos

R. Leveugle, K. Hadjiat, "Multi-Level Fault Injection Experiments Based on VHDL Descriptions: A Case Study", en Procs. 8th IEEE International On-Line Testing Workshop (IOLTW 2002), Isla de Bendor (Francia), pp. 107-111, Julio 2002.

Artículo citado: [IOLTW 2000].

M. Rebaudengo, M. Sonza Reorda, M. Violante, "Analysis of SEU Effects in a Pipelined Processor", en Procs. 8th IEEE International On-Line Testing Workshop (IOLTW 2002), Isla de Bendor (Francia), pp. 112-116, Julio 2002.

Artículo citado: [DFT 2001].

G.C. Cardarilli, F. Kaddour, A. Leandri, M. Ottavi, S. Pontarelli, R. Velazco, "Bit-Flip Injection in Processor-Based Architectures: A Case Study", en Procs. 8th IEEE International On-Line Testing Workshop (IOLTW 2002), Isla de Bendor (Francia), pp. 117-127, Julio 2002.

Artículo citado: [DFT 2001].

A. Castelnuovo, A. Fin, F. Fummi, F. Sforza, "Emulation-based Design Errors Identification", Procs. IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2002), pp. 365-371, Vancouver, Canadá, Noviembre 2002.

Artículo citado: [Journal of Systems Architecture].

Ö. Askerdal, "On Impact and Tolerance of Data Errors with Varied Duration in Microprocessors", Tesis Doctoral, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Suecia 2003.

Artículo citado: [DFT 2001].

A. Ademaj, "Assessment of Error Detection Mechanisms of the Time-Triggered Architecture Using Fault Injection", Tesis Doctoral, Institut für Technische Informatik, Real-Time Systems Group,Vienna University of Technology, Viena, Austria, Abril 2003.

Artículos citados: [DFT 2001, DFT 2002, PRDC2002].

R. Leveugle, L. Antoni, B. Fehér, "Dependability Analysis: A New Application for Run-Time Reconfiguration", en Procs. International Parallel and Distributed Processing Symposium (IPDPS'03), pp. 173b, Niza, Francia, Abril 2003.

Artículo citado: [DFT 2001].

A. Fin, F. Fummi, M. Poncino, G. Pravadelli, "A SystemC-based Framework for Properties Incompleteness Evaluation", en Procs. 4th International Workshop on Microprocessor Test and Verification Common Challenges and Solutions (MTV'03), pp. 89-94, Austin (Texas, EE.UU.), Mayo 2003.

Artículo citado: [DFT 2000].

A. Ejlali, S.G. Miremadi, H. Zarandi, G. Asadi, S.B. Sarmadi, "A Hybrid Fault Injection Approach Based on Simulation and Emulation Co-operation", en Procs. 2003 International Conference on Dependable Systems and Networks (DSN'03), pp. 479-488, San Francisco (California, EE.UU.), Junio 2003.

Artículo citado: [DFT 2001].

W. Müller, W. Rosenstiel, J. Ruf, "Systemc: Methodologies and Applications", Wolfgang Muller (Edt), Kluwer Academic Publishers, Julio 2003.

Artículo citado: [DFT 2001].

R. Leveugle, K. Hadjiat, "Multi-Level Fault Injections in VHDL Descriptions: Alternative Approaches and Experiments", Journal of Electronic Testing: Theory and Applications", Vol. 19, Issue 5, pp. 559-575, Octubre 2003.

Artículo citado: [DFT 2001].

A. Ammari, R. Leveugle, M. Sonza-Reorda, M. Violante, "Detailed Comparison of Dependability Analyses performed at RT and Gate Levels", en 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03), pp. 336-343, Boston, (Massachusetts, EE.UU.), Noviembre 2003.

Artículo citado: [DFT 2001].

M.Rebaudengo, M. Sonza Reorda, M. Violante, "Accurate Analysis of Single Event Upsets in a Pipelined Microprocessor", Journal of Electronic Testing: Theory and Application,Vol, 19, pp. 577-584, 2003.

Artículo citado: [IOLTW 2001].

H. Sivencrona, "On the Design and Validation of Fault Containment Regions in Distributed Communication Systems", Tesis Doctoral, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Suecia 2004.

Artículos citados: [DFT 2002, PRDC2002].

8.3 Trabajo futuro

A partir del trabajo desarrollado para la realización de la presente tesis, se pueden continuar diferentes líneas de investigación.

En relación con las técnicas de inyección, un punto primordial es la disminución de costes a la hora de aplicar las diferentes técnicas de inyección de fallos basadas en VHDL, especialmente el coste temporal a la hora de aplicar la técnica de los mutantes para inyectar fallos transitorios. Otro aspecto que queda abierto es la inclusión de los perturbadores y mutantes en una herramienta de inyección de fallos, de forma que su utilización esté automatizada.

Otra línea propuesta para mejorar el funcionamiento y las prestaciones de la herramienta de inyección de fallos (VFIT) es el diseño de una versión distribuida de la misma, la cual permitiría la utilización de una red de ordenadores para realizar las distintas simulaciones así como efectuar un análisis también distribuido. Una vez que todos los ficheros estén analizados, un *analizador central* fusionaría los resultados y se los proporcionaría al usuario. Obviamente, todo el proceso de distribución debe ser transparente al usuario.

Durante los experimentos de inyección de fallos, han surgido varias cuestiones. En primer lugar, un factor que hay que estudiar es la influencia de la carga de trabajo en los valores de la Confiabilidad, ya que parece demostrado que ésta influye en la manifestación de los fallos que puedan ocurrir en el sistema. Además, dentro de este campo, también queda abierto el estudio de la inyección de fallos en función de la frecuencia de paso del programa. Es decir, si tenemos en cuenta que un programa está compuesto por un conjunto de instrucciones, la inyección de fallos podría realizarse para perturbar aquellas instrucciones más ejecutadas durante la ejecución normal del mismo.

Otra cuestión está relacionada con los modelos de fallos. Al haber sido aumentados estos modelos, habría que estudiar más profundamente cómo influyen éstos en los resultados obtenidos durante los diferentes experimentos de inyección, incluso intentar comparar distintas técnicas de inyección de fallos que inyecten los mismos modelos. También relacionado con los modelos de fallos, está la representatividad de los mismos. Una línea pendiente sería el estudio de los fallos que se producen en las nuevas tecnologías submicrónicas y cómo se manifiestan en niveles de abstracción superiores (lógico, RT, algorítmico, sistema operativo y sistema).

En relación con la cuestión anterior, también queda abierto el estudio de la influencia de las diferentes técnicas de inyección de fallos en los resultados obtenidos.

Una vez comprobada la efectividad de la inyección de fallos basada en VHDL en sistemas distribuidos de tiempo real tolerantes a fallos, el siguiente paso sería la validación de otros sistemas de este tipo que se vayan a introducir en aviónica y automoción, como puede ser la arquitectura *FlexRay*.

Palabras Clave

Bibliografía

[Ademaj02] "A Methodology for Dependability Evaluation of the Time-Triggered Architecture Using Software Implemented Fault Injection"; Astrit Ademaj; Procs. 4th European Dependabe Computing Conference (EDCC-4), Published in Lecture notes in Computer Science Nº 2485, pp. 172-190, Springer-Verlag, Tolouse, Francia, Octubre 2002.

[Aftabjahani97] "Functional Fault Simulation of VHDL Gate Level Models"; S.A. Aftabjahani y Z. Navabi; Procs. 1997 VHDL International Users' Forum (VIUF'97), pp. 18-23, Arlington (Virginia, EE.UU.), Octubre 1997.

[Aidemark01] "GOOFI: Generic Object-Oriented Fault Injection Tool"; J. Aidemark, J. Vinter, P. Folkesson, J. Karlsson; Procs. 2001 International Conference on Dependable Systems and Networks (DSN 2001), pp. 83-88, Göteborg (Suecia), Julio 2001.

[Al-Hayek99] "From Design Validation to Hardware Testing: A Unified Approach"; G. Al-Hayek, C. Robach; Journal of Electronic Testing, 14(2):133-140, Febrero 1999.

[Alderighi03] "A Tool for Injecting SEU-like Faults into the Configuration Control Mechanism of Xilinx Virtex FPGAs"; M.Alderighi, F. Casini, S. D'Angelo, M. Mancini, A. Marmo, S. Pastore, G.R. Sechi; Procs. 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03), pp. 71-78, Boston, (Massachusetts, EE.UU.), Noviembre 2003.

[Amendola96] "Fault Behavior Observation of a Microprocessor System through a VHDL Simulation-Based Fault Injection Experiment"; A.M. Amendola, A. Benso, F. Corno, L. Impagliazzo, P. Marmo, P. Prineto, M. Rebaudengo, M. Sonza Reorda; in Procs. European Design Automation Conference with EURO-VHDL, (EURO-DAC with EURO-VHDL – EURO-DAC'96), Génova, Suiza, Septiembre 1996.

[Amendola97] "Experimental Evaluation of Computer-Based Railway Control Systems"; A.M. Amendola, L. Impagliazzo, P. Marmo, F. Poli; Procs. 27th International Symposium on Fault-Tolerant Computing (FTCS-27), pp. 380-384, Seattle (Washington, EE.UU.), Junio 1997.

[Amerasekera97] "Failure Mechanisms in Semiconductor Devices"; E.A. Amerasekera y F.N. Najm; 2nd edition, John Wiley & Sons, 1997.

[Antoni02] "Using Run-Time Reconfiguration for Fault Injection in Hardware Prototypes"; L. Antoni, R. Leveugle, B. Fehér; Procs. IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2002), pp. 405-413, Vancouver, Canadá, Noviembre 2002.

[Ashenden92] "The VHDL Cookbook", Peter Ashenden, University of Adelaide, South Australia, Technical Report, 1992.

[Arlat84] "Sur la Certification des systèmes informatiques: le projet EVE – Application au poste d'aiguillage informatisé"; J. Arlat, P. Blanquart, J.C. Laprie; *Actes 4ème Colloque International Fiabilité et Maintenabilité*, Perros-Guirec et Trégastel, pp. 650-656, Mayo 1984.

[Arlat90a] "Validation de la Sûreté de Fonctionnement par Injection de Fautes. Méthode – Mise en Oeuvre"; J. Arlat; Thése présentée à L'Institut National Polytechnique de Toulouse, Rapport de Recherche LAAS Nº 90-399; Diciembre 1990.

[Arlat90b] "Fault injection for dependability validation: a methodology and some applications"; J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.C. Fabre, J.C. Laprie, E. Martins, D. Powell; IEEE Transactions on Software Engineering, 16(2):166-182, Febrero 1990.

[Arlat92a] "Fault Injection for the Experimental Validation of Fault-Tolerant Systems"; J. Arlat; Procs. 1992 IEICE Workshop on Fault-Tolerant Systems, pp. 33-40, Kyoto, Japón, Junio 1992.

[Arlat92b] "Fault Injection and Dependability Evaluation of Fault-Tolerant Systems"; J. Arlat, A. Costes, Y. Crouzet, J.C. Laprie, D. Powell; Research Report, PDCS nº 52, Esprit Basic Research Action 3902 & Esprit Basic Research Project 6362, Enero 1992.

[Arlat93] "Fault Injection and Dependability Evaluation of Fault-Tolerant Systems"; J. Arlat, A. Costes, Y. Crouzet, J. Laprie, D. Powell; IEEE Transactions on Computers,.42(8):913-923, 1993.

[Arlat99] "Validation-Based Development of Dependable Systems"; J. Arlat, J. Boué, Y. Crouzet; IEEE Micro, pp. 66-79, Julio-Agosto 1999.

[Arlat03a] "MEFISTO: A Series of Prototype Tools for Fault Injection into VHDL Models"; J. Arlat, J. Boué, Y. Crouzet, E. Jenn, J. Aidemark, P. Folkesson, J, Karlsson, J. Ohlsson, M. Rimén; en Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation, Edited by Alfredo Benso and Paolo Prinetto, Kluwer Academic Publishers, pp. 177-193, Octubre 2003.

[Arlat03b] "Comparison of Physical and Software Implemented Fault Injection Techniques"; J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, G.H. Leber; IEEE Transactions on Computers,52(9):1115-1133, Septiembre 2003.

[Armstrong89] "Chip-Level Modelling with VHDL"; J. R. Armstrong; Prentice Hall, 1989.

[Armstrong92] "Test generation and Fault Simulation for Behavioral Models"; J.R. Armstrong, F.S. Lam, P.C. Ward; en Performance and Fault Modelling with VHDL. (J.M.Schoen ed.), pp.240-303, Englewood Cliffs, Prentice-Hall, 1992.

[Avresky92] "Fault injection for the formal testing of fault tolerance"; D. Avresky, J. Arlat, J.C. Laprie, Y. Crouzet; Procs. 22nd International Symposium on Fault-Tolerant Computing (FTCS-22), pp. 345-354, Boston (Massachusetts, EE.UU.), Julio 1992.

[Aylor90] "A Fundamental Approach to Uninterpreted/Interpreted Modeling of Digital Systems in a Common Simulation Environment"; J.H. Aylor, R.D. Williams, R. Waxman, B.W. Johnson, R.L. Blackburn; Technical Report 900724.0, University of Virginia, 1990.

[Baldini03] "BOND: An Agents-Based Fault Injector for Windows NT"; A. Baldini, A. Benso, P. Prinetto; en Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation, Edited by Alfredo Benso and Paolo Prinetto, Kluwer Academic Publishers, pp. 111-123, Octubre 2003.

[Baraza99] "Diseño de una Herramienta de Inyección de Fallos sobre Modelos en VHDL"; J.C. Baraza; Trabajo de Doctorado (6 créditos); Septiembre 1999.

[Baraza00] "A Prototype of a VHDL-Based Fault Injection Tool"; J.C. Baraza, J. Gracia, D. Gil, P.J. Gil; IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2000), pp. 396-404, Yamanashi, Japan, Octubre 2000.

[Baraza02] "A Prototype of a VHDL-Based Fault Injection Tool. Description and Application"; J.C. Baraza, J. Gracia, D. Gil, P.J. Gil; Journal of Systems Architecture, 47(10):847-867, 2002.

[Baraza03] "Contribución a la Validación de Sistemas Complejos Tolerantes a Fallos en la Fase de Diseño. Nuevos Modelos de Fallos y Técnicas de Inyección de Fallos"; J.C. Baraza; *Tesis doctoral*, Departamento de Informática de Sistemas y Computadores, Universidad Politécnica de Valencia, Octubre 2003.

[Bauer01] "Assumption Coverage under Different Failure Modes in the Time-Triggered Architecture"; G. Bauer, H. Kopetz, P. Puschner; Research Report Nº 14/2001, Vienna University of Technology, Viena, Austria, 2001.

[Benso98a] "A Hybrid Fault Injection Methodology for Real Time Systems"; A. Benso, P.L. Civera, M. Rebaudengo, M. Sonza Reorda, A. Ferro; FastAbstracts 28th International Symposium on Fault-Tolerant Computing (FTCS-28), pp. 74-75, Munich, Alemania, Junio 1998.

[Benso98b] "A Fault Injection Environment for Microprocessor-based Boards"; A. Benso, P. Prinetto, M. Rebaudengo, M. Sonza Reorda; Procs. International Test Conference 1998 (ITC'98), pp. 768-773, Washington (D.C., EE.UU.), Octubre 1998.

[Benso98c] "EXFI: A Low Cost Fault Injection System for Embedded Microprocessor-Based Boards"; A. Benso, P. Prinetto, M. Rebaudengo, M. Sonza Reorda; ACM Transactions on Design Automation of Electronic Systems, 3(4):626-634, Octubre 1998.

[Benso98d] "Fault-List Collapsing for Fault-Injection Experiments"; A. Benso, M. Rebaudengo, L. Impagliazzo, P. Marmo; Procs. 1998 Annual Reliability and Maintainability Symposium (RAMS98), pp. 383-388, Anaheim (California, EE.UU.), Enero 1998.

[Benso99a] "A Low-Cost Programmable Board for Speeding-Up Fault Injection in Microprocessor-Based Systems"; A. Benso, P.L. Civera, M. Rebaudengo, M. Sonza Reorda; Procs. 1999 Annual Reliability and Maintainability Symposium (RAMS99), pp. 171-177, Washington (D.C., EE.UU.), Enero 1999.

[Benso99b] "FlexFi: A Flexible Fault Injection Environment for Microprocessor-Based Systems"; A. Benso, M. Rebaudengo, M. Sonza Reorda; Procs. 18th International Conference on Computer Safety, Reliability and Security (SAFECOMP'1999), pp. 323-335, Toulouse, Francia, Septiembre 1999.

[Berrojo02] "New Techniques for Speeding-up Fault Injection Campaigns"; L. Berrojo, I. González, F. Corno, M. Sonza Reorda, G. Squillero, L. Entrena, C. López; Procs. 2002 Design Automation and Test in Europe (DATE 2002), pp. 847-852, París, Francia, Marzo 2002.

[Berwanger01] "FlexRay – the communication system for advanced automotive control systems"; J. Berwanger, C. Ebner, A. Schedl, R. Belschner, S. Fluhrer, P. Lohrmann, E. Fuchs, D. Millinger, M. Sprachmann, F. Bogenberger, G. Hay, A. Krueger, M. Rausch, W.O. Budde, P. Fuhrmann, R. Mores; SAE 2001 World Congress, Society of Automotive Engineers, Detroit, (Michigan, EE.UU.), artículo nº 2001-01-0676, Abril 2001.

[Blanc01] "Stratified Fault Injection using Hardware and Software-implemented Tools"; S. Blanc, J.C. Campelo, P.J. Gil, J.J. Serrano; Procs. of 4th IEEE International Workshop on Desing and Diagnostics of Electronic Circuits and Systems (DDECS'01), pp. 259-266, Gyor, Hungría, Abril 2001.

[Blanc02a] "Three Different Fault Injection Techniques Combined to Improve the Detection Efficiency for Time-Triggered Systems"; S. Blanc, P.J. Gil, A. Ademaj, H. Sivencrona, J. Torin; Procs. 5th IEEE International Workshop on Desing and Diagnostics of Electronic Circuits and Systems (DDECS'02), pp. 412-415, Brno, República Checa, Abril 2002.

[Blanc02b] "A Fault Hypothesis Study on the *TTP/C* using VHDL-based and Pin-level Fault Injection Techniques"; S. Blanc, J. Gracia, P.J. Gil; Procs. IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2002), pp. 254-262, Vancouver, Canadá, Noviembre 2002.

[Blanc04] "Experiences during the Experimental Validation of the Time-Triggered Architecture"; S. Blanc, J. Gracia, P.J. Gil; Procs. Design, Automation and Test in Europe (DATE'04), pp. 30256-30261, CNIT la Défense, París, Febrero 2004.

[Bosch92] "CAN Specification, version 2.0"; Robert Bosch GmbH, 1992.

[Boué96] "Verification of Fault Tolerance by Means of Fault Injection into VHDL Simulation Models"; J. Boué, J. Arlat, Y. Crouzet, P. Pétillon; LAAS Report nº 96-463. Diciembre 1996.

[Boué97] "Early Experimental Verification of Fault Tolerance: the VHDL-based Fault Injection tool Mefisto-L"; J. Boué, P. Pétillon, Y. Crouzet, J. Arlat; DeVa ESPRIT Project 20072 Second Year Report; 1997.

[Boué98] "MEFISTO-L: A VHDL-Based Fault Injection Tool for the Experimental Assessment of Fault Tolerance"; J. Boué, P. Pétillon, Y. Crouzet; Procs. 28th International Symposium on Fault-Tolerant Computing (FTCS-28), pp. 168-173. Munich, Alemania, Junio 1998.

[Bouricius69] "Reliability modeling techniques for self-repairing computer systems"; W. Bouricius, W. Carter, P. Schneider; Procs. 24th ACM National Conference, pp. 295-309, 1969.

[Burgun96] "Serial Faul Emulation"; L. Burgun, F. Reblewski, G. Fenelon, J. Bariber, O. Lepape; Procs. 1996 Design Automation Coference (DAC'96), pp. 801-806, Las Vegas (Nevada, EE.UU.), Junio 1996.

[Byteflight01] "Technical Specification 100.38, Byteflight – Interface – IC"; Revisión del 09.07.2001.

[Calvez93] "Embedded Real-Time Systems"; J.P. Calvez; John Wiley & Sons, 1993.

[Campelo99] "Diseño y Validación de Nodos de Proceso Tolerantes a Fallos de Sistemas Industriales Distribuidos"; José Carlos Campelo; *Tesis doctoral*, Departamento de Informática de Sistemas y Computadores, Universidad Politécnica de Valencia, Junio 1999.

[Cardarilli02] "Bit flip Injection in Processor-based Architectures: A Case Study"; G.C. Cardarilli, F. Kaddour, A. Leandri, M. Ottavi, S. Pontarelli, R. Velazco; Procs. 8th International On-Line Testing Workshop (IOLTW'02), pp. 117-127, Isla de Bendor, Francia, Julio 2002.

[Carreira95] "Xception: a Technique for the Experimental Evaluation of Dependability in Modern Computers"; J. Carreira, H. Madeira, J.G. Silva; Procs. 5th Working Conference on Dependable Computing for Critical Applications (DCCA-5), pp. 135-148, Urbana-Champaign (Illinois, EE.UU.), Setiembre 1995.

[Carreira98] "Xception: a Technique for the Experimental Evaluation of Dependability in Modern Computers"; J. Carreira, H. Madeira, J.G. Silva; IEEE Transactions on Software Engineering, 24(2):125-136, Febrero 1998.

[Celeiro96] "VHDL Fault Simulation for Defect-Oriented Test and Diagnosis of Digital ICs"; F. Celeiro, L. Dias, J. Ferreira, M.B. Santos, J.P. Teixeira; Procs. European Design Automation Conference with EURO-VHDL, (EURO-DAC with EURO-VHDL – EURO-DAC'96), Geneva, Suiza, September 16-20, 1996.

[Cha93] "A fast and accurate gate-level transient fault simulation environment"; H. Cha, E. M. Rudnick, G.S. Choi, J.H. Patel, R.K. Iyer; Procs. 23rd Symposium on Fault-Tolerant Computing Systems (FTCS-23), Toulouse, Francia, pp. 310-319, Junio 1993.

[Cha96] "A Gate-Level Simulation Environment for Alpha-Particle-Induced Transient Faults"; H. Cha, E.M. Rudnick, J.H. Patel, R.K. Iyer, G.S. Choi; IEEE Transactions on Computes, 45(11):1248-1256, Noviembre 1996.

[Chen93] "Testing and evaluating fault tolerant protocols by deterministic fault injection"; M.E.Y. Chen; Fortschritt-Beritche, nº 260, VDI Verlag, Düsseldorf, Alemania, Julio 1993.

[Cheng95] "Fault Emulation: A New Approach to Fault Grading"; K.T. Cheng, S.Y. Huang, W.J. Dai; Procs. 1995 International Conference on Computer-Aided Design (ICCAD'95), pp. 681-686, 1995.

[Cheng99] "Fault Emulation: A New Methodology for Fault Grading"; K.T. Cheng, S.Y. Huang, W.J. Dai; IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 18(10):1487-1495, Octubre 1999.

[Choi92] "FOCUS: An experimental environment for fault sensitivity analysis"; G.S. Choi, R.K. Iyer; IEEE Transactions on Computers, 41(12):1515-1526, Diciembre 1992.

[Choi93] "Wear-out simulation environment for VLSI designs"; G.S. Choi, R.K. Iyer; Procs. 23rd International Symposium on Fault-Tolerant Computing (FTCS-23), pp. 320-329, Toulouse, Francia, Junio 1993.

[Civera01a] "FPGA-Based Fault Injection Techniques for Fast Evaluation of Fault Tolerance in VLSI Circuits"; P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante; Procs. 11th International Conference Field Programmable Logic and Applications (FPL2001), pp. 493-502, Belfast, Reino Unido, Agosto 2001.

[Civera01b] "FPGA-Based Fault Injection for Microprocessor Systems"; P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante; Procs. 10th Asian Test (ATS 2001), pp. 304-312, Kyoto, Japón, Noviembre 2001.

[Civera01c] "Exploiting Circuit Emulation for Fast Hardness Evaluation"; P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante; IEEE Transactions on Nuclear Science, 48(6):2210-2216, Diciembre 2001.

[Civera01d] "Exploiting FPGA-based Techniques for Fault Injection Campaigns on VLSI Circuits"; P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante; Procs. 2001 International Symposium on Defect and Fault Tolerance (DFT'01), pp. 250-258, San Francisco (California, EE.UU.), Octubre 2001.

[Clark92] "REACT: Reliable Architecture Characterization Tool"; J.A. Clark, D.K. Pradhan; Technical Report TR-92-CSE-22, University of Massachusetts, Junio 1992.

[Clark95] "Fault Injection. A method for validating computer-system dependability"; J.A. Clark, D.K. Pradhan; IEEE Computer; 28(6):47-56, Junio 1995.

[Constantinescu01] "Dependability Analysis of a Fault-Tolerant Processor"; C. Constantinescu; Procs. 2001 Pacific Rim International Symposium on Dependable Computing (PRDC 2001), pp. 63-67, Seúl, Corea del Sur, Diciembre 2001.

[Constantinescu02] "Impact of Deep Submicron Technology on Dependability of VLSI Circuits"; C. Constantinescu; Procs. International 2002 Conference on Dependable Systems and Networks (DSN'02), pp. 205-209, Washington (D.C., EE.UU.), Junio 2002.

[Coppens98] "VHDL Modelling and Analysis of Fault Secure Systems"; J. Coppens, D. Al-Khalili, C. Rozon; Procs. 1998 Design, Automation and Test in Europe (DATE 1998), pp. 148-152, París, Francia, Marzo 1998.

[Corno00] "RT-level Fault Simulation Techniques based on Simulation Command Scripts"; F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero; Procs. XV Conference on Design of Circuits and Integrated Systems (DCIS 2000), pp. 825-830, Montpellier, FranciaNoviembre 2000.

[Corno03] "New Acceleration Techniques for Simulation-Based Fault Injection"; F. Corno, L. Entrena, C. López, M. Sonza Reorda, G. Squillero; en Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation, Edited by Alfredo Benso and Paolo Prinetto, Kluwer Academic Publishers, pp. 217-232, Octubre 2003.

[Courtois92] "Sûreté de Fonctionnement Informatique. Evolutions 1987-1992. Tendances et Perspectives"; B. Courtois, M.C. Gaudel, J.C. Laprie, D. Powell; Rapport rédigé à la demande de la Direction Centrale de la Qualité du CNES, Diciembre 1992.

[Czeck90] "Effects of transient gate-level faults on program behaviour"; E.W. Czeck, D.P. Siewiorek; Procs. 20th International Symposium on Fault-Tolerant Computing (FTCS-20), pp. 236-243, Newcastle Upon Tyne, Reino Unido, Junio 1990.

[Damm86] "The effectiveness of software error detection mechanisms in real time operating systems"; A. Damm; Procs. 16th International Symposium on Fault-Tolerant Computing (FTCS-16), pp. 171-176, Viena, Austria, Julio 1986.

[Damm88] "Experimental Evaluation of error detection and self checking coverage of components of a distributed real-time systems"; A. Damm; Tesis doctoral, Universität Wien, Viena, Austria, Octubre 1988.

[Dawson96a] "Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection"; S. Dawson, F. Jaharnian, T. Mitton, T.L. Tung; Procs. 26th International Symposium on Fault-Tolerant Computing (FTCS-26), pp. 404-414, Sendai, Japón, Junio 1996.

[Dawson96b] "ORCHESTRA: A Fault Injection Environment for Distributed Systems"; S. Dawson, F. Jaharnian, T. Mitton; Technical Report CSE-TR-318-96, University of Michigan, Electrical Engineering and Computer Science Department (EECS), 1996.

[DBENCH01] "State of the Art", Deliverable CF1 del proyecto Dependability Benchmarking (DBench), IST-2000-25245, Agosto 2001.

[DBENCH02] "Fault Representativeness", Deliverable ETIE2 of Dependability Benchmarking Project (DBench), IST-2000-25245, Julio 2002.

[DBENCH03] "Dependability Benchmarking (DBench)", Páginas web del proyecto Dependability Benchmarking (DBench), IST-2000-25425, en http://www.laas.fr/DBench y http://www.cordis.lu/esprit/#Projects.

[deAndrés03] "Reconfiguración Dinámica de FPGAs para la aceleración de la Inyección de Fallos Basada en Simulación"; D. de Andrés, J. Albaladejo, L. Lemus; III Jornadas Sobre Computación Reconfigurable y Aplicaciones (JCRA2003), pp. 39-46, Madrid, España, Septiembre 2003.

[DeLong94] "A Novel Fault Injection Technique for Behavioural-Level Modeling using VHDL"; T.A. DeLong, B.W. Johnson, J.A. Profeta III, D. Bozzolo; VHDL International Users' Forum Fall Conference, Noviembre 1994.

[DeLong96a] "A Fault Injection Technique for VHDL Behavioural-Level Models"; T.A. DeLong, B.W. Johnson, J.A. Profeta III; IEEE Design & Test of Computers, 13(4): 24-33, Winter 1996.

[DeLong96b] "Simulator Independent Fault Simulation Using WAVES"; T.A. DeLong, D.T. Smith, B.W. Johnson; Procs. 1996 VHDL International Users' Forum (VIUF'96), pp 129-138, Durham, (Carolina del Norte), EE.UU., Octubre 1996.

[DeMillo87] "Software Testing and Evaluation"; R.A. De Millo, W.M. McCracken, R. Martin, J.F. Passafiume; The Benjamin/Cummings Pub. Company, 1987.

[Dewey92] "VHDL: Toward a Unified View of Design"; A. Dewey, A.J. de Geus; IEEE Design and Test of Computers, 9(2):8-17, Abril/Junio 1992.

[DGil98a] "Fault Injection into VHDL Models: Analysis of the Error Syndrome of a Microcomputer System"; D. Gil, J.C. Baraza, J.V. Busquets, P. Gil; Procs. 24th EUROMICRO Conference, Vol. 1, pp. 418-425. Västerås, Suecia, Agosto 1998.

[DGil98b] "Using VHDL in the Techniques of Fault Injection based on Simulation"; D. Gil, J.V. Busquets, J.C. Baraza, P. Gil; Procs. XIII Design of Circuits and Integrated Systems Conference (DCIS'98), pp. 174-180, Madrid, España, Noviembre 1998.

[DGil99a] "Validación de Sistemas Tolerantes a Fallos mediante Inyección de Fallos en Modelos VHDL"; Daniel Gil Tomás; *Tesis doctoral*, Departamento de Informática de Sistemas y Computadores, Universidad Politécnica de Valencia, Julio 1999.

[DGil99b] "Fault Injection into VHDL Models: Experimental Validation of a Fault Tolerant Microcomputer System"; D. Gil, R. Martínez, J.V. Busquets, J.C. Baraza, P. Gil; Procs. 3rd European Dependable Computing Conference (EDCC-3), pp. 191-208; Praga, Republica Checa; Septiembre 1999.

[DGil00] "A Study of the effects of Transient Fault Injection into the VHDL Model of a Fault-Tolerant Microcomputer System"; D. Gil, J. Gracia, J.C. Baraza, P.J. Gil; Procs. 6th IEEE International On-Line Testing Workshop (IOLTW'2000), pp. 73-79, Palma de Mallorca, España, Julio 2000.

[DGil03a] "Study, Comparison and Application of different VHDL-Based Fault Injection Techniques for the Experimental Validation of a Fault-Tolerant System"; D. Gil, J. Gracia, J.C. Baraza, P.J. Gil, Microelectronics Journal, Special Section on Defect and Fault Tolerance in VLSI Systems, 34(1):41-51, Enero 2003.

[DGil03b] "VHDL Simulation-Based Fault Injection Techniques"; D. Gil, J.C. Baraza, J. Gracia, P.J. Gil; en Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation, Edited by Alfredo Benso and Paolo Prinetto, Kluwer Academic Publishers, pp. 159-176, Octubre 2003.

[Dilger97] "Towards an Architecture for Safety Related Fault Tolerant Systems in Vehicles"; E. Dilger, L.A. Johanson, H. Kopetz, M. Krug, P. Liden, G. McCall, P. Mortara, B. Müller, U. Panizza, S. Poledna, A. Schedl, J. Söderberg, N. Strömberg, T. Thurner; Procs. of the European Conference on Safety and Reliability (ESREL'97), pp. 1021-1030, Portugal, Junio 1997.

[Dugan89] "Coverage modelling for dependability analysis of fault-tolerance systems"; J.B. Dugan, K.S. Trivedi; IEEE Transactions on Computers, 38(6):775-787, Junio 1989.

[Dupuy90] "NEST: A Network Simulation and Prototyping Testbed"; A. Dupuy, J. Schwartz, Y. Yemini, D. Bacon; Communications of the ACM, 33(10):64-74, Octubre 1990.

[Echtle91] "Evaluation of deterministic fault injection for fault tolerant protocol testing"; K. Echtle, Y. Chen; Procs. 21st International Symposium on Fault-Tolerant Computing (FTCS-21), pp. 418-425, Montreal, Canadá, Junio 1991.

[Echtle92] "The EFA fault injector for fault tolerant distributed system testing"; K. Echtle, M. Leu; Procs. IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, pp. 28-35, Amherst, EE.UU., Julio 1992.

[Ejlali03] "A Hybrid Fault Injection Approach Based on Simulation and Emulation Co-operation"; A. Ejlali, S.G. Miremadi, H. Zarandi, G. Asadi, S.B. Sarmadi; Procs. 2003 International Conference on Dependable Systems and Networks (DSN'03), pp. 479-488, San Francisco (California, EE.UU.), Junio 2003.

[Fang93] "A Mechanism for Gate Oxide Damage in Nonuniform Plasmas"; S. Fang, J. McVittie; Procs. 31st International Reliability Physics Symposium (IRPS '93), pp. 13-17, Atlanta (Georgia, EE.UU.), Marzo 1993.

[FAST01] "Integrating Formal Approaches to Specification, test Case Generation and Automatic Design Verification (FAST)", Página web del proyecto FAST ESPRIT 25581, en http://www.prover.temp.pi.se/fast.

[Fabre99] "Assessment of COTS microkernel by fault injection"; J.C. Fabre, F. Sallés, M. Rodríguez, J. Arlat; Procs. 7th IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-7), pp. 19-38, San Jose, EE.UU., Enero 1999.

[Finelli87] "Characterisation of fault recovery through fault injection on FTMP"; G.B. Finelli; IEEE Transactions on Reliability, 36(2):164-170, Junio 1987.

[FIT-A99] "Proposal: Fault Injection for TTA", proposal nº IST-1999-10748, documento A,1999.

[FIT-B99] "Fault Injection for *TTA* (FIT). IST-1999-10748", proposal nº IST-1999-10748, documento B, 1999.

[FIT-C99] "Fault Injection for *TTA* (FIT). IST-1999-10748", proposal nº IST-1999-10748, documento C, 1999.

[FIT00a] "Fault Injection for the TTA. Report Deliverable I, Part I", Octubre 2000.

[FIT00b] "Fault Injection for the TTA. Report Deliverable I, Part II", Octubre 2000.

[FIT01a] "FIT – Fault Injection for the TTA. Deliverable 2"; Carinthia Tech Institute, TU Vienna, TTTech, Czech Technical University, Universidad Politécnica de Valencia, Chalmers University, Motorola, Volvo; Informe público, 6 de Junio del 2001.

[FIT01b] "Fault Injection for the TTA. Report Deliverable III", Agosto 2001.

[FIT02a] "FAULT INJECTION FOR *TTA* – FIT. Deliverable 5.1 – 5.5: Combined Report, based on the Consolidated Status Report – Measurements", Diciembre 2002.

[FIT02b] "FAULT INJECTION FOR *TTA* – FIT. Deliverable 6: Architecture Critique·", Octubre 2002.

[FIT02c] "Fault Injection for *TTA* (FIT)", Páginas web del proyecto Fault Injection on *TTA* (FIT), IST-1999-10748, en http://www3.cti.ac.at/fit y http://www.cordis.lu/esprit/#Projects.

[FIT02d] "FAULT INJECTION FOR *TTA* – FIT. Fault Injection Techniques – a management summary", Septiembre 2002.

[Folkesson98] "A comparison of simulation based and scan chain implemented fault injection"; P. Folkesson, S. Svensson, J. Karlsson; in Procs. 28th International Symposium on Fault-Tolerant Computing (FTCS-28), pp. 284-293, Munich, Alemania, Junio 1998.

[Folkesson99] "Assessment and Comparison of Physical Fault Injection Techniques"; P. Folkesson; Thesis for the degree of Doctor of Philosophy, Department of Computer Engineering, Chalmers University of Technology, Göteborg (Suecia), 1999.

[Fuchs96] "An Evaluation of the Error Detection Mechanisms in MARS Using Software-Implemented Fault Injection"; E. Fuchs; Procs. 2nd European Dependable Computing Conference (EDCC-2), pp. 73-90, Taormina, Italia, Octubre 1996.

[Führer00] "Time-Triggered Communication on CAN (Time Triggered CAN – TTCAN)"; T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, M. Walther; 7th International CAN Conference 2000, Amsterdam, Holanda, Octubre 2000.

[Gaisler97] "Evaluation of a 32-bit Microprocessor with Built-in Concurrent Error Detection"; J. Gaisler; Procs. 27th International Symposium on Fault-Tolerant Computing (FTCS-27), pp. 42-47, Seattle (Washington, EE.UU.), Junio 1997.

[Gaisler02] "A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8 Architecture"; J. Gaisler; Procs. 2002 International Conference on Dependable Systems and Networks (DSN 2002), pp. 409-415, Bethesda (Maryland, EE.UU.), Junio 2002.

[Galiay80] "Physical Versus Logical Fault Models MOS LSI Circuits: Impact on Their Testability"; J. Galiay, Y. Crouzet, M. Vergniault; IEEE Transactions on Computers, 29(6):527-531, Junio 1980.

[Ghosh91] "On behavior fault modeling for digital design"; S. Ghosh, T.J. Chakraborty; Journal of Electronic Testing: Theory and Applications, nº 2, pp. 135-151, 1991.

[Ghosh95a] "Fault Injection in the Design Process Using VHDL"; A.K. Ghosh, T.A. DeLong, B.W. Johnson, J.A. Profeta III; VHDL International Users' Forum Fall Conference, Octubre 1995.

[Ghosh95b] "Fault Injection for Logic Synthesis Design Using VHDL"; Anup K. Ghosh, Todd A. DeLong, Barry W. Johnson, Joseph A. Profeta III; Mentor Graphics User's Group, 12th Annual International Conference, Portland Marriot, Portland (OR), October 23-27, 1995.

[Ghosh95c] "System-Level Modeling in the ADEPT Environment of a Distributed Computer System for Real-Time Applications"; A.K. Ghosh, B.W. Johnson, J.A. Profeta III; Procs. 1995 International Computer Performance and Dependability Symposium (IPDS-95), pp. 194-203, Erlangen, Alemania, Abril 1995.

[Goswami92] "DEPEND: A Simulation-Based Environment for System Level Dependability Analysis"; K.K. Goswami, R.K. Iyer; Technical Report CRHC 92-11, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana (Illinois, EE.UU.), Junio 1992.

[Goswami93] "Simulation of software behaviour under hardware faults"; K.K. Goswami, R.K. Iyer; Procs. 23rd International Symposium on Fault-Tolerant Computing (FTCS-23), pp. 340-347, Toulouse, Francia, Junio 1993.

[Goswami97] "DEPEND: A Simulation-Based Environment for System Level Dependability Analysis"; K.K. Goswami, R.K. Iyer, L. Young; IEEE Transactions on Computers, 46(1):60-74, Enero 1997.

[Gracia00a] "Application Of Different VHDL-Based Fault Injection Techniques to the Validation of a Fault-Tolerant Microcomputer System"; J. Gracia, D. Gil, J. C. Baraza, P. J. Gil; FastAbstracts of the International Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8 – DSN'2000), pp. B-54 - B-55, New York, (NY, EE.UU.), Junio 2000.

[Gracia00b] "Inyección de fallos basadas en nuevas técnicas de simulación de modelos VHDL"; J. Gracia; Trabajo de doctorado de 10 créditos; Valencia, Septiembre 2000.

[Gracia01a] "A Study of the Experimental Validation of Fault Tolerant Systems Using Different VHDL-Based Fault Injection Techniques"; J. Gracia, J.C. Baraza, D. Gil, P.J. Gil; Procs. of the 7th IEEE On-Line Testing Workshop (IOLTW'2001), pp. 140, Taormina, Italia, Julio 2001.

[Gracia01b] "Comparison and Application of different VHDL-Based Fault Injection Techniques"; J. Gracia, J.C. Baraza, D. Gil, P.J. Gil; Procs. 2001 IEEE International. Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2001), pp. 233-241, San Francisco (EE.UU.), Octubre 2001.

[Gracia02a] "Studying Hardware Fault Representativeness with VHDL Models"; J. Gracia, D. Gil, L. Lemus, P.J. Gil; XVII Conference on Design of Circuits and Integrated Systems (DCIS 2002), pp. 33-38, Santander, Noviembre 2002.

[Gracia02b] "Using VHDL-Based Fault Injection to exercise Error Detection Mechanisms in the Time-Triggered Architecture"; J. Gracia, D. Gil, J.C. Baraza, P.J. Gil; Procs. 2002 Pacific Rim International Symposium on Dependable Computing (PRDC 2002), Tsukuba, Japón, pp. 316-320, Diciembre 2002.

[Gracia02c] "VFIT: Una herramienta automática para la inyección de fallos en VHDL"; J. Gracia, S. Blanc, J.C. Baraza, D. Gil, P.J. Gil; Actas Seminario Anual de Automática, Electrónica Industrial e Instrumentación (SAAEI'02), Alcalá de Henares, Madrid, España, pp. II-289–II-292, Septiembre 2002.

[Gracia03a] "Early Diagnosis of Hard Real-Time Fault-Tolerant Embedded Systems"; J. Gracia, J.C. Baraza, D. Gil, P.J. Gil; Procs. 6th International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2003), pp.157-164, Poznań, Polonia, Abril 2003.

[Gracia03b] "Using VHDL-Based Fault Injection for the Early Diagnosis of a *TTP/C* Controller"; J. Gracia, J.C. Baraza, D. Gil, P.J. Gil; Transactions on Information and Systems, Special Issue on Dependable Computing, The Institute of Electronics, Information and Communication Engineers, Vol. E86-D, Nº 12, pp. 2634-2641, Diciembre 2003.

[Gunneflo89] "Evaluation of error detection schemes using fault injection by heavy-ion radiation"; U. Gunneflo, J. Karlsson, J. Torin; Procs. 19th International Symposium on Fault-Tolerant Computing (FTCS-19), pp. 218-227, Chicago (Illinois, EE.UU.), Junio 1989.

[Gunneflo90] "The effects of power supply disturbances on the MC6809E microprocessor"; U. Gunneflo; Technical Report 89, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Suecia, 1990.

[Güthoff95] "Combining Software-Implemented and Simulation-Based Fault Injection into a Single Fault Injection Method"; J. Güthoff, V. Sieh; Procs. 25th Annual International Symposium on Fault Tolerant Computing (FTCS-25), Pasadena (California, EE.UU.), Junio 1995.

[Han94] "DOCTOR: An integrated software fault injection environment"; S. Han, H.A. Rosenberg, K.G. Shin; Procs. 3rd IEEE International Workshop on Integrating Error Models with Fault Injection, Annapolis, EE.UU., Abril 1994.

[Hawkins00] "CMOS IC Failure Mechanism and Defect Based Testing"; C. Hawkins; 2nd Summer Course on Selected Microelectronic Design & Test Topics, Palma de Mallorca, Julio 2000.

[Hayne99] "Behavioral Fault Modeling in a VHDL Synthesis Environment"; R.J. Hayne, B.W. Johnson; Procs. 17th IEEE VLSI Test Symposium; pp. 333-340, San Diego (California, EE.UU.), Abril 1999.

[Hedenetz98] "Fault Injection and Fault Modeling for a Safety Critical Automotive Communication System"; B. Hedenetz, A.V. Schedl; Procs. of the European Conference on Safety and Reliability (ESREL'98), Trondheim, Noruega, Junio 1998.

[Herout02] "Model-Based Dependability Evaluation Method for *TTP/C* Applications"; P. Herout, S. Racek, J. Hlavička; Procs. 4th European Dependable Computing Conference (EDCC-4), Published in Lecture notes in Computer Science nº 2485, pp. 271-282, Springer-Verlag, Toulouse, Francia, Octubre 2002.

[Hlavièka96] "Functional Validation of Fault-Tolerant Asynchronous Algorithms"; J. Hlavièka, S. Racek, P. Šmrha; Procs. EUROMICRO-22, pp. 143-150, Praga, República Checa, Septiembre 1996.

[Hong96] "An FPGA-Based Hardware Emulator for Fast Fault Emulation"; J.H. Hong, S.A. Hwang, C.W. Wu; Procs. 1996 Midwest Symposium on Circuit and Systems, Ames (Iowa, EE.UU.), Agosto 1996.

[Hoyme92] "SAFEbus™"; K. Hoyme, K. Driscoll; 11th AIAA/IEEE Digital Avionics Systems Conference, pp. 68-73, Seattle, (Washington, EE.UU.), Octubre 1992.

[Höxer02] "UMLinux – A Tool for Testing a Linux System's Fault Tolerance"; H.J. Höxer, K. Buchacker, V. Sieh; Procs. LinuxTag 2002, Karlsruhe, Alemania, Junio 2002.

[Hu99] "A Unified Gate Oxide Reliability Model"; C. Hu, Q. Lu; Procs. 39th International Reliability Physics Symposium (IRPS '99), pp. 47-52, San Diego (California, EE.UU.), Marzo 1999.

[Hsueh97] "Fault Injection Techniques and Tools"; M. Sueh, T. Tsai, R.K. Iyer; IEEE Computer, 20(4):75-82, Abril 1997.

[Hwang98] "Sequential Circuit Fault Simulation Using Logic Emulation"; S.A. Hwang, J.H. Hong, C.W. Wu; IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 17(8):724-736, Agosto 1998.

[IEEE93] IEEE Standard VHDL Language Reference Manual; IEEE Std 1072-1993.

[Impagliazzo03] "Development of a Hybrid Fault Injection Environment. The Birth and Growth of LIVE"; L. Impagliazzo, F. Poli; en Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation, Edited by Alfredo Benso and Paolo Prinetto, Kluwer Academic Publishers, pp. 81-93, Octubre 2003.

[Iyer86] "A measurement-based model for workload dependence of CPU errors"; R.K. Iyer, D. Rosseti; IEEE Transactions on Computers, 35(6):511-519. Junio 1986.

[Iyer95] "Experimental Evaluation"; R.K. Iyer; Procs. 25th International Symposium on Fault-Tolerant Computing (FTCS-25) – Special Issue, pp. 115-132, Pasadena (California, EE.UU.), Junio 1995.

[Jagannath95] "Impact of hardware and software faults on ARQ schemes – An experimental study"; A. Jagannath, S. Rai; Procs. Annual Reliability and Maintainability Symposium, pp. 479-485, Washington (EE.UU.), Enero 1995.

[Jenn92] "Mise en Oeuvre de l'Injection de Fautes en VHDL"; E. Jenn, J. Arlat; LAAS Report nº 92-66. Julio 1992.

[Jenn93a] "Fault Injection into VHDL Models for the Validation of Fault-Tolerant Systems"; E. Jenn, J. Arlat, Y. Crouzet; LAAS Report nº 93-030. Febrero 1993.

[Jenn93b] "Fault Injection into VHDL Models: The MEFISTO Tool"; E. Jenn, J. Arlat, M. Rimén, J. Ohlsson, J. Karlsson; LAAS Report nº 93-460. Diciembre 1993.

[Jenn94a] "Sur la validation des systémes tolérant les fautes: injection de fautes dans de modéles de simulation VHDL"; Eric Jenn; Thése; Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS (LAAS); LAAS Report nº 94-361; 1994.

[Jenn94b] "Fault injection into VHDL models: the MEFISTO tool"; E. Jenn, J. Arlat, M. Rimén, J. Ohlsson, J. Karlsson; Procs. 24th Int. Symposium on Fault-Tolerant Computing (FTCS-24), pp. 356-363. Austin, (Texas, EE.UU.), Junio 1994.

[Jones87] "Line Width Dependence of Stresses in Aluminium Interconnect"; R.E. Jones; Procs. 25th International Reliability Physics Symposium (IRPS'87), pp. 9-14, San Diego (California, EE.UU.), Abril 1987.

[Kanawati92] "FERRARI: A tool for the validation of system dependability properties"; G.A. Kanawati, N.A. Kanawati, J.A. Abraham; Procs. 22nd International Symposium on Fault-Tolerant Computing (FTCS-22), pp. 336-344, Boston (Massachusetts, EE.UU.), Julio 1992.

[Kanawati95] "FERRARI: A flexible software based fault and error injection system"; G.A. Kanawati, N.A. Kanawati, J.A. Abraham; IEEE Transactions on Computers, 44(2):248-260, Febrero 1995.

[Kanoun89] "Croissance de la Sûrété de fonctionement des logiciels. Caracterisation – Modelisation – Evaluation"; K. Kanoun; Thése présentée a L'Institut National Polytechnique de Toulouse, Septiembre 1989.

[Kao93] "FINE: a fault injection and monitoring environment for tracing UNIX system behaviour under faults"; W. Kao. R.K. Iyer, D. Tang; IEEE Transactions on Software Engineering, 19(11):1105-1118, Noviembre 1993.

[Kao94] "DEFINE: a distributed fault injection and monitoring environment"; W. Kao, R.K. Iyer; Workshop on Fault Tolerant Parallel and Distributed Systems, Junio 1994.

[Karlsson89] "Use of heavy-ion radiation from 252-californium for fault injection experiments"; J. Karlsson, U. Gunneflo, J. Torin; Procs. 1st International Working Conference on Dependable Computing for Critical Applications (DCCA-1), Santa Barbara (EE.UU.), Agosto 1989.

[Karlsson90] "Transient fault effects in the MC6809E 8-bit microprocessor: A comparison of results of physical and simulated fault injection experiments"; J. Karlsson; Technical Report 96, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Suecia, 1990.

[Karlsson94] "Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms"; J. Karlsson, P. Lidén, P. Dahlgren, R. Johansson, U. Gunneflo; IEEE Micro, pp. 8-23, Febrero 1994.

[Karlsson95] "Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture", J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber and J. Reisinger, 5th IFIP Working Conference on Dependable Computing for Critical Applications (DCCA-5), pp. 267-287, Urbana Champaign, (Illinois, EE.UU.), Septiembre, 1995

[Kilty95] "VHDL/VITAL Fault Simulation"; P. Kilty; Procs. 1995 VHDL Users Group/VHDL International Users' Forum – Fall Conference (VUG/VIUF FALL 95), Boston (Massachusetts, EE.UU.), Octubre 1995.

[Kopetz82] "The Architecture of MARS"; H. Kopetz, F. Lohnert, W. Merker, G. Pauthner, Technische Universität Berlin, Technical Report MA 82/2, Abril 1982.

[Kopetz94] "TTP – A Protocol for Fault-Tolerant Real-Time Systems"; H. Kopetz, G. Grünsteidl, IEEE Computer, Vol. 27(1):14-23, Enero 1994.

[Kopetz97] "Real-Time Systems. Design Principles for Distributed Embedded Applicatiosn"; H. Kopetz; Kluwer Academic Publishers, 1997.

[Kopetz98a] "The Time-Triggered Architecture"; H. Kopetz; Procs. 1st International Symposium on Object-Oriented Real-Time Distributed Computing, Kyoto, Japan, Abril 1998.

[Kopetz98b] "The Time-Triggered Architecture"; H. Kopetz; IFIP International Workshop on Dependable Computing and its Applications (DCIA98), Johannesburg, South Africa, Enero 1998.

[Kopetz99] "*TTP/C* Protocol"; H. Kopetz, TTTech 1999. Disponible en http://www.ttpforum.org

[Kopetz01] "A Comparison of *TTP/C* and FlexRay"; H. Kopetz; Research Report 10/2001, University of Technology, Viena, Austria, Mayo 2001.

[Kopetz03] "The Time-Triggered Architecture"; H. Kopetz, G. Bauer; Proceedings of the IEEE, 91(1):112-126, Enero 2003.

[Kumar94] "ADEPT: A Unified System Level Modeling Design Environment"; S. Kumar, R.H. Klence, J.H. Aylor, B.W. Johnson, R.D. Williams, R. Waxman; Procs. 1st Rapid Prototyping of Application Specific Signal Processors Conference (RASSP'94), pp. 114-123, Arlington (Virginia, EE.UU.), Agosto 1994.

[Lajolo00] "Evaluating System Dependability in a Co-Design Framework"; M. Lajolo, M. Rebaudengo, M. Sonza Reorda, M. Violante, L. Lavagno; Procs. 2000 Design, Automation and Test in Europe (DATE 2000), pp. 586-590, París, Francia, Marzo 2000.

[Lala83] "Fault detection, isolation, and reconfiguration in FTMP: Methods and Experimental Results"; J.H. Lala; Procs. 5th AIAA/IEEE Digital Avionics Systems Conference, pp. 21.3.1-21.3.9, 1983.

[Laprie85] "Dependable Computing and Fault-Tolerance: Concepts and Terminology"; J.C. Laprie; Procs. 15th IEEE International Symposium on Fault-Tolerant Computing (FTCS-15), pp. 2-11, Ann Arbor (Michigan, EE.UU.), Junio 1985.

[Laprie92] "Dependability, Basic Concepts and Terminology"; J.C. Laprie; Springer-Verlag, 1992.

[Leveugle00a] "Fault Injection in VHDL Descriptions and Emulation"; R. Leveugle; Procs. 2000 International Symposium on Defect and Fault Tolerance (DFT'00), pp. 414-420, Yamanashi, Japón, Octubre 2000.

[Leveugle00b] "Optimized Generation of VHDL Mutants for Injection of Transition Errors"; R. Leveugle, K. Hadjiat; Procs. 13th Symposium on Integrated Circuits and Systems Design (SBCCI'00), pp. 243-248, Manaus, Brasil, Septiembre 2000.

[Leveugle01a] "A Low-Cost Hardware Approach to Dependability Validation of IPs"; R. Leveugle; Procs. 2001 International Symposium on Defect and Fault Tolerance (DFT'01), pp. 242-249, San Francisco (California, EE.UU.), Octubre 2001.

[Ley01] "AS8202. TTP/C-C2 Controller. Functional Description"; M. Ley, TTChip Gmbh, Documento Nº D-032-S-10-008, Publicado el 1 de Agosto del 2001.

[Li95] "Cellular Automata for Efficient Parallel Logic and Fault Simulation"; Y.L. Li, C.W. Wu; IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 14(6):740-749, Junio 1995.

[Li97] "VLSI Design of a Cellular-Automata Based Logic and Fault Simulator"; Y.L. Li, Y.C. Lai, C.W. Wu; Procs. 1997 National Science Council Part A: Physical Science and Engineering, 21:189-199, Taipei (Taiwan, China), Mayo 1997.

[Lima01] "On the Use of VHDL Simulation and Emulation to Derive Error Rates"; F. Lima, S. Rezgui, L. Carro, R. Velazco, R. Reis; Procs. 6th European Conference on Radiation and its Effects on Components and Systems (RADECS 2001), Grenoble, Francia, Septiembre 2001.

[LIS96] "Guide de la Sûreté de Fonctionnement"; Laboratoire d'Ingenierie de la Sûreté de fonctionnement (LIS); Cépaduès – Éditions; 1996.

[Lomelino86] "Error Propagation in a Digital Avionic Processor – A Simulation-Based Study"; D. Lomelino, R.K. Iyer; Procs. 1986 Real-Time Systems Symposium (RTSS'86), pp. 218-225, New Orleans (Louisiana, EE.UU), Diciembre 1986.

[López98a] "A method to perform error simulation in VHDL"; C. López, T. Riesgo, Y. Torroja, E. De la Torre, J. Uceda; Procs. XIII Design of Circuits and Integrated Systems Conference (DCIS 1998), pp. 495-500, Madrid, España, Noviembre 1998.

[López98b] "An Error Simulation to Estimate the Quality of Design Validation Experiments"; C. López, T. Riesgo, Y. Torroja, E. De la Torre, J. Uceda; Procs. Forum on Design Language (FDL'98), Lausanne, Suiza, Septiembre 1998.

[Lovric93] "ProFI: Processor fault injection for dependability validation"; T. Lovric, K. Echtle; Procs. International Workshop on Fault and Error Detection for Dependability Validation of Computer Systems, Göteborg, Suecia, Junio 1993.

[Madeira94] "RIFLE: A general purpose pin-level fault injector"; H. Madeira, M. Rela, F. Moreira, J.G Silva; Procs. 1st European Dependable Computing Conference (EDCC-1), pp 199-216, Berlín, Alemania, Octubre 1994.

[Madritsch01] "Fault Injection for the Time-Triggered Architecture (FIT)", edited by C. Madritsch, Supplement of the 2001 Int. Conference on Dependable Systems and Networks (DSN 2001), Special Track: European Dependability Initiative, Göteborg, Suecia, pp. D-25 - D-27, Julio 2001.

[Maier02] "Time-Triggered Architecture: A Consistent Computing Platform"; R. Maier, G. Bauer, G. Stöger, S. Poledna; IEEE Micro, 22(4):36-45, Julio-Agosto 2002.

[Majzik98] "Dependability Analysis in the HIDE Framework"; I. Majzik, A. Bondavalli; HIDE Document, Junio 1998.

[Martinet92] "Contribution à l'Evaluation de l'Efficacité du Test Fonctionnel de Microprocesseurs"; B. Martinet; Thèse de Doctorat de troisième cycle, Institut National Politechnique de Grenoble, Noviembre 1992.

[Martínez99] "Experimental Validation of High-Speed Fault-Tolerant Systems Using Physical Fault Injection"; J. R. Martínez, P.J. Gil, G. Martín, C. Pérez, J.J. Serrano; Procs. 7th International Working Conference on Dependable Computing for Critical Applications (DCCA-7), pp. 233-249, 1999.

[Maxion93] "Detection and discrimination of injected network faults"; R.A. Maxion, R.T. Olszewski; Procs. 23rd International Symposium on Fault-Tolerant Computing (FTCS-23), pp. 198-207, Toulouse, Francia, Junio 1993.

[McVittie96] "Plasma Charging Damage: An Overview"; J. McVittie; Procs. 1st International Symposium on Plasma Process-Induced Damage (P2ID), pp. 7-20, Santa Clara (California, EE.UU.), Mayo 1996.

[Miczo90] "VHDL as a Modeling-for-Testability Tool"; A. Miczo; Procs. 35th Computer Society International Conference (COMPCON'90), pp.403-409, San Francisco (California, EE.UU.), Febrero-Marzo 1990.

[Miner93] "Verification of Fault-Tolerant Clock Synchronization Systems"; P.S. Miner; NASA Techincal Paper 3349, NASA Langley Research Center, Hampton, VA, Noviembre 1993.

[Miner00] "Analysis of the SPIDER fault-tolerance protocols"; P.S. Miner; en C. Michael Holloway, editor, 5th NASA Langley Formal Methods Workshops (LFM 2000), Junio 2000.

[Miremadi92] "Two software techniques for on line error detection"; G. Miremadi, J. Karlsson, U. Gunneflo, J. Torin; Procs. 22nd International Symposium on Fault-Tolerant Computing (FTCS-22), pp. 328-335, Boston (Massachusetts, EE.UU.), Julio 1992.

[Miremadi95] "Evaluating processor-behaviour and three error detection mechanisms using physical fault-injection"; G. Miremadi, J. Torin; IEEE Transactions on Reliability, 44(3):441-454, Setiembre 1995.

[Model98] Model Technology, "ModelSim EE/PLUS Reference Manual", 1998.

[Model01a] Model Technology, "ModelSim SE/User's Manual, Version 5.5e", 24 Septiembre 2001.

[Model01b] Model Technology, "ModelSim SE Command Reference. Version 5.5e", 2001.

[Nexus99] The Nexus Forum™ Standard for a Global Embedded Processor Debug Interface. IEEE-ISTO 5001-1999, en http://www.ieee-isto.org/Nexus5001, 1999.

[Oates93] "Electromigration in Stress-Voided Al Alloy Metallizations for Submicron IC Technologies"; A. Oates; Procs. 31st International Reliability Physics Symposium (IRPS '93), pp. 297-303, Atlanta (Georgia, EE.UU.), Marzo 1993.

[Ohlsson92] "A Study of the Effect of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog"; J. Ohlsson, M. Rimén, U. Gunneflo; Procs. 22nd Int. Symposium on Fault-Tolerant Computing (FTCS-22), pp. 316-325, Boston, EE.UU., Julio 1992.

[Parreira03] "A Novel Approach to FPGA-Based Hardware Fault Modeling and Simulation"; A. Parreira, J.P. Teixeira, M. Santos; Procs. 6th International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2003), pp.17-24, Poznań, Polonia, Abril 2003.

[Parrotta00a] "New Techniques for Accelerating Fault Injection in VHDL descriptions"; B. Parrotta, M. Rebaudengo, M. Sonza Reorda, M. Violante; Procs. 6th IEEE International On-Line Testing Workshop (IOLTW'2000), pp. 61-66, Palma de Mallorca, España, Julio 2000.

[Parrotta00b] "Speeding-up Fault Injection Campaigns in VHDL models"; B. Parrotta, M. Rebaudengo, M. Sonza Reorda, M. Violante; 19th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2000), Rotterdam, Holanda, pp. 27-36, Octubre 2000.

[Perry94] "VHDL"; D.L. Perry, McGraw-Hill, 1994.

[Pfeifer99] "Formal verification for time-triggered clock synchronization"; H. Pfeifer, D. Schwier, F.W. von Henke; En Charles B. Weinstock and John Rushby (eds), Dependable Computing for Critical Applications – 7, Vol. 12 de Dependable Computing and Fault Tolerant Systems, IEEE Computer Society, pp. 207–226, San Jose, (California, EE.UU.), Enero 1999.

[Pfeifer00] "Formal verification of the *TTA* group membership algorithm"; H. Pfeifer; en Tommaso Bolognesi and Diego Latella (eds), Formal Description Techniques and Protocol Specification, Testing and Verification, Procs. Of FORTE XIII/PSTV XX 2000, Kluwer Academic Publishers, pp. 3-18, Pisa, Italia, Octubre 2000.

[PGil92] "Sistema Tolerante a Fallos con Procesador de Guardia: Validación mediante Inyección Física de Fallos"; P.J. Gil; Tesis doctoral, Departamento de Ingeniería de Sistemas, Computadores y Automática (DISCA), Universidad Politécnica de Valencia, Septiembre 1992.

[PGil96] "Garantía de Funcionamiento: Conceptos Básicos y Terminología"; Pedro J. Gil; Informe Interno; Departamento de Ingeniería de Sistemas, Computadores y Automática (DISCA), Universidad Politécnica de Valencia, 1996.

[PGil97] "High speed fault injector for safety validation of industrial machinery"; P.J. Gil, J.C. Baraza, D. Gil, J.J. Serrano; Procs. 8th European Workshop of Dependable Computing (EWDC-8): Experimental validation of dependable systems, Göteborg, Suecia, Abril 1997.

[PGil03] "Pin-level Hardware Fault Injection Techniques"; P.J. Gil, S. Blanc, J.J. Serrano; capítulo 2.1 del libro "Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation", A. Benso y P. Prinetto ed. Book series: Frontiers in Electronic Testing, vol. 23, pp.63-80, Kluwer Academic Press, Octubre 2003.

[Pol96] "Short Loop Monitoring of Metal Step Coverage by Simple Electrical Measurements"; J.A. van der Pol, E.R. Ooms, H.T. Brugman; Procs. 34th International Reliability Physics Symposium (IRPS'96), pp. 148-155, Dallas (Texas, EE.UU.), Abril-Mayo 1996.

[Powell88] "Delta_4: Overall System Specification"; Delta_4 Consortium; D. Powell ed., Diciembre 1988.

[Powell99] "GUARDS: A Generic Upgradable Architecture for Real-Time Dependable Systems"; D. Powell, J. Arlat, L. Beus-Dukic, A. Bondavalli, P. Coppola, A. Fantechi, E. Jenn, C. Rabéjac, A. Wellings.; IEEE Transactions on Parallel and Distributed Systems, 10(6):580-599, Junio 1999.

[Powell01] "A Generic Fault-Tolerant Architecture for Real-Time Dependable Systems", D. Powell, Ed., Kluwer Academic Publishers, 2001.

[Pradhan86] "Fault-Tolerant Computing Theory and Techniques"; D.K. Pradhan; Prentice-Hall, 1986.

[Pradhan96] "Fault-Tolerant Computer System Design"; D.K. Pradhan; Prentice-Hall; 1996.

[Proteus96] "Probestar DVT-100 Application Note"; Proteus Corporation; Marzo 1996.

[Randell95] "Predictably Dependable Computing Systems", B. Randell, J.C. Laprie, H. Kopetz, B. Littlewood, Eds., ESPRIT Basic Research Series, Springer-Verlag, 1995.

[Reisinger94] "The design of a fail-silent processing node for MARS"; J. Reisinger, A. Steininger; Distributed Systems Engineering Journal, 1994.

[Reisinger95] "The PDCS implementation of MARS hardware and software"; J. Reisinger, A. Steininger, G. Leber; Predictably Dependable Computing Systems pp. 209-224, Springer Verlag, 1995.

[Riesgo96] "A Fault Model for VHDL Descriptions at the Register Transfer Level"; T. Riesgo, J. Uceda; Procs. European Design Automation Conference with EURO-VHDL, (EURO-DAC with EURO-VHDL – EURO-DAC'96), Geneva, Suiza, September 16-20, 1996.

[Rimén92a] "Validation of Fault Tolerance by Fault Injection in VHDL Models"; M. Rimén, J. Ohlsson, J. Karlsson, E. Jenn, J. Arlat; LAAS Report nº 92-469, Diciembre 1992.

[Rimén92b] "A Study of the Error Behavior of a 32-bit RISC Subjected to Simulated Fault Injection"; M. Rimén, J. Ohlsson; Procs. IEEE International Test Conference; pp. 696-704, Baltimore (Maryland, EE.UU.), Septiembre 1992.

[Rimén93a] "Design Guidelines of a VHDL-based Simulation Tool for the Validation of Fault Tolerance"; M. Rimén, J. Ohlsson, J. Karlsson, E. Jenn, J. Arlat; Procs. 1st Open Workshop ESPRIT Basic Research Project 6362: PDCS-2; pp. 461-483; Toulouse, Francia, 1993.

[Rimén93b] "A study of the error behaviour of a 32 bit RISC subjected to simulated transient fault injection"; M. Rimén, J. Ohlsson; PFCS2 Report, pp. 445-460, Septiembre 1993.

[Rimén97] "MEFISTO: Multilevel Error and Fault Injection Simulation Tool. User's Manual"; M. Rimén, J. Ohlsson, S. Svensson; Chalmers University of Technology, Göteborg, Suecia, 1997.

[Robach03] "Simulation-Based Fault Injection and Testing using the Mutation Technique"; C. Robach, M. Scholive; en Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation, Edited by Alfredo Benso and Paolo Prinetto, Kluwer Academic Publishers, pp. 195-215, Octubre 2003.

[Rodder95] "A Scaled 1.8V, 0.18μm Gate Lenght CMOS Technology: Device Design and Reliability Considerations"; M. Rodder, S. Aur, C. Chen; Technical Digest International Electron Devices Meeting (IEDM), pp. 415-418, Washington (D.C., EE.UU.), Diciembre 1995.

[Rodríguez98] "Assessment of COTS Microkernel-based System by Fault Injection and Fault Containment with Wrappers"; M. Rodríguez; Proyecto Final de Carrera, Facultad de Informática de la Universidad Politécnica de Valencia – LAAS-CNRS, Junio 1998.

[Rosenberg93] "Software fault injection and its application in distributed systems"; H.A. Rosenberg, K.G. Shin; Procs. 23rd International Symposium on Fault-Tolerant Computing (FTCS-23), pp. 208-217, Toulouse, Francia, Junio 1993.

[Rushby99] "Systematic formal verification for fault-tolerant time-triggered algorithms"; J. Rushby; IEEE Transactions on Software Engineering, 25(5):651–660, Septiembre/Octubre 1999.

[Rushby01a] "A Comparison of Bus Architectures for Safety-Critical Embedded Systems"; J. Rushby; CSL Technical Report, Septiembre 2001.

[Rushby01b] "Bus Architectures for Safety-Critical Embedded Systems"; J. Rushby; 1st Workshop on Embedded Software (EMSOFT 2001), Springer-Verlag Lecture Notes in Computer Science, Vol 2211, pp. 306-323, Lago Tahoe, (California, EE.UU.), Octubre 2001.

[Rushby01c] "Formal verification of transmission window timing for the time-triggered architecture"; J. Rushby; Technical Report, Computer Science Laboratory, SRI International, Menlo Park, (California, EE.UU.), Marzo 2001.

[Rushby02a] "An overview of formal verification for the time-triggered architecture"; J. Rushby; Procs. the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'02), Springer-Verlag, Lecture Notes in Computer Science, Vol. 2469, pp. 83–105, Oldenburg, Alemania, Septiembre 2002.

[SAE-C94] "Class C application requirements-J2056/1", SAE Handbook, SAE Press, 1994, Disponible en http://www.sae.org.

[Saito93] "Reliability Improvement in Blanket Tungsten CVD Contact Filling Process for High Aspect Ratio Contact"; T. Saito, H. Aoki, T. Tamaru, N. Owada; Procs. 31st International Reliability Physics Symposium (IRPS'93), pp. 334-339, Atlanta (Georgia, EE.UU.), Marzo 1993.

[Samson97] "Validating Fault Tolerant Designs Using Laser Fault Injection (LFI)"; J.R. Samson Jr., W. Moreno, F.J. Falquez; Procs. 1997 International Wokshop on Defect and Fault Tolerance in VLSI Systems (DFT'97), pp. 175-183, París, Francia, Octubre 1997.

[Samson98] "A Technique for Automated Validation of Fault Tolerant Designs Using Laser Fault Injection (LFI)"; J.R. Samson Jr., W. Moreno, F.J. Falquez; Procs. 28th International Symposium on Fault-Tolerant Computing (FTCS-28), pp. 162-167, Munich, Alemania, Junio 1998.

[Santos01] "eXception: An Evaluation Tool towards the Demanding Availability of Networking Products", N.D. Santos, D. Costa; FastAbstracts 2001 International Conference on Dependable Systems and Networks (DSN 2001), Göteborg, Suecia, Julio 2001.

[Schmid82] "Upset exposure by means of abstraction verification"; M.E. Schmid, R.L. Trapp, A.E. Davidoff, G.M. Masson; Procs. 12th International Symposium on Fault-Tolerant Computing (FTCS-12), pp. 237-244, Santa Mónica, EE.UU., Junio 1982.

[Schuette86] "Experimental evaluation of two concurrent error detection schemes"; M. Schuette, J. Shen, D. Siewiorek, Y. Zhu; Procs. 16th International Symposium on Fault-Tolerant Computing (FTCS-16), pp. 128-143, Viena, Austria, Julio 1986.

[Schwartz83] "Specifying and Verifying Ultra-Reliability and Fault-Tolerance Properties"; R.L. Schwartz, P.M. Melliar-Smith; Procs. 26th Computer Society International Conference (COMPCON'83), pp. 71-76, San Francisco (California, EE.UU.), Marzo 1983.

[Schwier98] "Mechanical Verification of Clock Synchronization Algorithms"; D. Schwier, F.W. von Henke; en Formal Techniques in Real-Time and Fault-Tolerant Systems, Vol. 1486 de Lecture Notes in Computer Science, pp. 262-271, Lyngby, Dinamarca, Septiembre 1998.

[Segall88] "FIAT–Fault Injection based Automated Testing Environment"; Z. Segall, D. Vrsalovic, D. Soewoprek, D. Yaskin, J. Kownavki, J. Barton, D. Rancey, A. Robinson, T. Lin; Procs. 18th International Symposium on Fault-Tolerant Computing (FTCS-18), pp. 102-107, Tokio, Japón, Junio 1988.

[Shaw01] "Accurate CMOS Bridge Fault Modeling With Neural Network-Based VHDL Saboteurs"; D.B. Shaw, D. Al-Khalili, C.N. Rozon; Procs. 2001 International Conference on Computer-Aided Design (ICCAD'01), pp. 531-536, San Jose (California, EE.UU.), Noviembre 2001.

[Shaw03] "IC Bridge Fault Modeling for IP Blocks Using Neural Network-Based VHDL Saboteurs"; D.B. Shaw, D. Al-Khalili, C.N. Rozon; IEEE Transactions on Computers, 52(10):1285-1297, Octubre 2003.

[Shen85] "Inductive Fault Analysis of MOS Integrated Circuits"; J.P. Shen, W. Maly, F.J. Ferguson; IEEE Design and Test of Computers, 2(4):13-26, Diciembre 1985.

[Shivakumar02] "Modeling the Effect of Technology Trends on Soft Error Rate of Combinational Logic"; P. Shivakumar, M. Kistler, S. Keckler, D. Burger, L. Alvisi; Procs. 2002 International Conference on Dependable Systems and Networks (DSN 2002), pp. 389-402, Washington (D.C., EE.UU.), Junio 2002.

[Sieh93] "Fault Injector using UNIX ptrace Interface"; V. Sieh; Internal Report nº 11/93, Noviembre 1993.

[Sieh96] "VHDL-based Fault Injection with VERIFY"; V. Sieh, O. Tschäche, F. Balbach; Internal Report Nº 5/96, IMMD III, University of Erlangen-Nürnberg, Agosto 1996.

[Sieh97a] "Comparing Different Fault Models Using Verify"; V. Sieh, O. Tschäche, F. Balbach; Procs 6th Dependable Computing for Critical Applications Working Conference (DCCA-6), Grinau, Alemania, Marzo 1997.

[Sieh97b] "VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions"; V. Sieh, O. Tschäche, F. Balbach; Procs. 27th Annual International Symposium on Fault Tolerant Computing (FTCS-27), pp. 32-36, Seatle, (Washington, EE.UU.), Junio 1997.

[Sieh97c] "System Dependability Analysis using VHDL Models with Integrated Fault Descriptions"; V. Sieh, O. Tschäche, F. Balbach; Procs. 8th European Workshop on Dependable Computing (EWDC-8), Göteborg, Suecia, 1997.

[Shie98] "System Dependability Analysis using VHDL Models with Integrated Fault Descriptions"; V. Sieh, O. Tschäche, F. Balbach; Procs. 8th European Workshop on Dependable Computing (EWDC-8), Göteborg, Suecia, 1998.

[Sieh02] "Testing the Fault-Tolerance of Networked Systems"; V. Sieh, K. Buchacker; Procs. 2002 International Conference on Architecture of Computing Systems (ARCS 2002), pp. 95-105, Karlsruhe (Alemania), Abril 2002.

[Siewiorek82] "The Theory and Practice of Reliable System Design"; D.P. Siewiorek, R.S. Swarz; Digital Press, Bedford (Massachusetts, EE.UU.), 1982.

[Siewiorek92] "Reliable Computer Systems. Design and Evaluation"; D.P. Siewiorek, R.S. Swarz; 2ª ed., Digital Press, 1992.

[Siewiorek94] "Reliable Computer Systems. Design and Evaluation"; D. P. Siewiorek; 3rd ed., Digital Press. 1994.

[Sivencrona00] "Design Principles for Dependable Time-Triggered Control Systems"; H. Sivencrona, J. Hedberg, O. Bridal; Technical Report, PALBUS Task 10.7, Diciembre 2000.

[Sivencrona03] "Heavy-ion Fault Injection in the Time-triggered Communication Protocol"; H. Sivencrona, P. Johannessen, M. Persson, J. Torin; Procs. 1st Latin American Symposium on Dependable Computing (LADC 2003), pp. 69-80, São Paulo, Brasil, Lecture notes 2847, Springer-Verlag, Berlin Heidelberg, Octubre 2003.

[Smith96] "System Dependability Evaluation via a Fault List Generation Algorithm"; D.T. Smith, B.W. Johnson, J.A. Profeta III; IEEE Transactions on Computers, 45(8):974 - 979, Agosto 1996.

[Sprachmann98] "Communication Controller. Functional Design"; Edited by Michael Sprachmann; Authors: O. Maischberger, D. Maurer, M. Sprachmann, J. Vilanek, Version 1.1.2, 22.01.98, Informe Interno.

[ST99] "M29F200BT, M29F200BB. 2 Mbit (256Kb x8 or 128Kb x16, Boot Block). Single Supply Flash Memory"; STMicroelectronics, Octubre 1999.

[Stathis01] "Physical and Predictive Models of Ultra Thin Oxide Reliability in CMOS Devices and Circuits"; J.H. Stathis; Procs. 39th International Reliability Physics Symposium (IRPS '01), pp. 132-150, Orlando (Florida, EE.UU.), Abril-Mayo 2001.

[Steininger99] "Economic Online Self-Test in the Time Triggered Architecture"; A. Steininger, C. Temple; IEEE Design & Test of Computers, 16(3):81-89, Julio-Septiembre 1999.

[Stewart97] "Board Level Automated Fault Injection for Fault Coverage and Diagnosis Efficiency"; B.A. Stewart; Procs. International Test Conference (ITC'97), pp. 649-654, Washington (D.C., EE.UU.), Noviembre 1997.

[Stott97] "Dependability Analysis of a Commercial High-Speed Network"; D.T. Stott, M.C. Hsueh, G.L. Ries, R.K. Iyer; Procs. 27th International Symposium on Fault-Tolerant Computing (FTCS-27), pp. 248-257, Seattle (Washington, EE.UU.), Junio 1997.

[Stott98] "Dependability Analysis of a High-Speed Network Using Software-Implemented Fault Injection and Simulated Fault Injection"; D.T. Stott, G.L. Ries, M.C. Hsueh, R.K. Iyer; IEEE Transactions on Computers, 47(1):108-119, Enero 1998.

[Stott00] "NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors"; D.T. Stott, B. Floering, D. Burke, Z. Kalbarczyk, R.K. Iyer; Procs. 4th International Computer Performance and Dependability Symposium (IPDS-2K), pp. 91-100, Chicago (Illinois, EE.UU.), Marzo 2000.

[Sweet95] "Boeing's seventh wonder"; W. Sweet, D. Dooling; IEEE Spectrum, 32(10):20-23, Octubre 1995.

[Sylvester99] "Rethinking Deep-Submicron Circuit Design"; D. Sylvester, K. Keutzer; IEEE Computer, 32(11):25-33, Noviembre 1999.

[Tezaki90] "Measurement of Three Dimensional Stress and Modeling of Stress-Induced Migration Failure in Aluminium Interconnects"; A. Tezaki, T. Mineta, H. Egawa, T. Noguchi; Procs. 28th International Reliability Physics Symposium (IRPS '90), pp. 221-229, New Orleans (Louisiana, EE.UU.), Marzo 1990.

[Tsai95a] "FTAPE: a fault injection tool to measure fault tolerance"; T.K. Tsai, R.K. Iyer; Procs. 10th AIAA Computing in Aerospace, pp. 339-346, San Antonio (Texas, EE.UU.), Marzo 1995.

[Tsai95b] "Measuring fault tolerance with the FTAPE fault-injection tool"; T.K. Tsai, R.K. Iyer; Procs. 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation – Performance Tools '95 – 8th GI/ITG Conference on Measuring, Modeling and Evaluating Computing and Communication Systems – MMB'95, pp. 26-40, Heidelberg, Alemania, Septiembre 1995.

[TTP/C99] "*TTP/C* C1 Controller. Specification of the *TTP/C* C1 Controller", TTTech Computertechnik GmbH, disponible en http://www.tttech.com, Febrero 1999.

[Turner82] "A New Failure Mechanism: Al-Si Bond Pad Whisker Growth During Lifetest"; T. Turner, R.D. Parsons; IEEE Transactions on Component, Hybrids and Manufacturing Technology, 5:431-435, 1982.

[Vargas99a] "Design of SEU-Tolerant Processors for Radiation-Exposed Systems"; F. Vargas, A. Amory; 4th Annual IEEE International Workshop on High Level Design Validation and Test (HLDVT'99), pp. 110-116, San Diego, (California, EE.UU.), Noviembre 1999.

[Vargas99b] "Testability Verification of Embedded Systems Based on Weak Mutation Analysis"; F. Vargas, E. Bezerra, A. Terroso; 3rd International Workshop on Testing Embedded Core-Based Systems Chips (TECS'99), Dana Point, (California, EE.UU.), Abril 1999.

[Vargas00a] "Estimating Circuit Fault-Tolerance by Means of Transient-Fault Injection in VHDL"; F. Vargas, A. Amory, R. Velazco; Procs. 6th IEEE International On-Line Testing Workshop (IOLTW'2000), pp. 67-72, Palma de Mallorca, España, Julio 2000.

[Vargas00b] "Fault-Tolerance in VHDL Description: Transient-Fault Injection & Early Reliability Estimation"; F. Vargas, A. Amory, R. Velazco; 1st Latin-American Test Workshop (LATW'00), pp. 29-35, Rio de Janeiro, Brasil, Marzo 2000.

[Velazco90] "Failure Coverage of Functional test methods: a comparative experimental evaluation"; R. Velazco, C. Bellon, B. Martinet: Procs. 1990 International Test Conference (ITC'90), Washington (D.C., EE.UU.), Septiembre 1990.

[Velazco00a] "Transient Bitflip Injection in Microprocessor Embedded Applications"; R. Velazco, S. Rezgui; Procs. 6th International On-Line Testing Workshop (IOLTW'2000), pp. 80-84, Palma de Mallorca, España, Julio 2000.

[Velazco00b] "Predicting Error Rate for Microprocessor-based Digital Architectures through C.E.U. (Code Emulating Upsets) Injection"; R. Velazco, S. Rezhui, R. Ecoffet; IEEE Transactions on Nuclear Science, 47(6):2405-2411, Diciembre 2000.

[Velazco01a] "Upset-like Fault Injection in VHDL Descriptions: A Method and Preliminary Results"; R. Velazco, R. Leveugle, O. Calvo; TIMA Research Report ISRN TIMA-RR-01/10-6-FR, TIMA Laboratory, 2001.

[Velazco01b] "Upset-like Fault Injection in VHDL Descriptions: A Method and Preliminary Results"; R. Velazco, R. Leveugle, O. Calvo; Procs. 2001 IEEE Int. Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2001), pp. 259-267, San Francisco, EE.UU., Octubre 2001.

[VFIT02] "VFIT User Guide. Revision 1.0", Grupo de Sistemas Tolerantes a Fallos, Universidad Politécnica de Valencia, Julio 2002.

[Walker85] "A Model of Design Representation and Synthesis"; R.A. Walker, D.E. Thomas; Procs. 22nd ACM/IEEE Design Automation Conference (DAC '85), pp. 453-459, Las Vegas (Nevada, EE.UU.), Junio 1985.

[Walker00] "Modelling the Wiring of Deep Submicron ICs"; M.G. Walker; IEEE Spectrum, 27(3):65-71, Marzo 2000.

[XbyWire98] "X-By-Wire. Safey Related Fault Tolerant Systems in Vehicles"; Project nº 95/1329, Contract nº BRPR-CT95-0032, Document nº XbyWire-DB-6/6-24, Version 2.0.0, 26-11-1998.

[Young92] "A Hybrid Monitor Assisted Fault Injection Environment"; L.T. Young, R.K. Iyer, K.K. Goswami, C. Alonso; Procs. 3rd Dependable Computing for Critical Applications Conference (DCCA-3), pp. 281-302, Ed. Springer-Verlag, Palermo, Italia, Septiembre 1992.

[Yount96] "A methodology for the rapid injection of transient hardware errors"; C. Yount, D.P. Siewiorek; IEEE Transactions on Computers, 45(8):881-891, Agosto 1996.

[Yu03] "Fault Injection Techniques. A Perspective on the State of Research"; Y. Yu, B.W. Johnson; en Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation, Edited by Alfredo Benso and Paolo Prinetto, Kluwer Academic Publishers, pp. 7-39, Octubre 2003.

[Yuste99] "TTA. Time-Triggered Architecture"; P. Yuste; Trabajo de doctorado, Septiembre 1999.

[Yuste03] "INERTE: Integrated NExus-based Real-Time fault injection tool for Embedded systems"; P. Yuste, D. de Andrés, L. Lemus, J.J. Serrano, P.J. Gil; Procs. 2003 International Conference on Dependable Systems and Networks (DSN 2003), pág. 669, San Francisco (California, EE.UU.), Junio 2003.

[Zarandi03] "Dependability Analysis Using a Fault Injection Tool Based on Synthesizability of HDL Models"; H.R. Zarandi, S.G. Miremadi, A. Ejlali; Procs. 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03), pp. 485-492, Boston, (Massachusetts, EE.UU.), Noviembre 2003.