UNIVERSIDAD POLITÉCNICA DE VALENCIA

DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

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

Tesis Doctoral

Presentada por

D. Juan Carlos Baraza Calvo

Dirigida por

Dr. D. Pedro Joaquín Gil Vicente

Dr. D. Daniel Antonio Gil Tomás

Valencia, 2003

A Lucía, Alberto e Inés.

Eureka!

Arquímedes de Siracusa (287 a.C. – 212 a.C.)

Índice

Resumen

En el diseño de sistemas informáticos (y en particular, de aquéllos en los que, por las características del servicio que prestan, un mal funcionamiento puede provocar pérdida de vidas humanas, perjuicio económico, suspensión de servicios primordiales, etc.), se establece como prioridad esencial conseguir que funcionen correctamente durante el mayor tiempo posible y con un elevado nivel de eficacia.

Los sistemas que regulan servicios críticos disponen de unos mecanismos especiales que les proporcionan una cierta inmunidad a la ocurrencia de averías que puedan causar un cese o deterioro del servicio prestado. Por ello, se les denomina Sistemas Tolerantes a Fallos, o STF.

Se define el concepto de Confiabilidad como un conjunto de funciones (o atributos) que permiten cuantificar la calidad del servicio prestado en cuanto a averías producidas, y en consecuencia, el grado de confianza que el usuario puede depositar en el sistema. Al desarrollar cualquier sistema tolerante a fallos es preciso validarlo, o lo que es lo mismo, cuantificar sus parámetros de Confiabilidad.

Entre los numerosos métodos y técnicas existentes para validar sistemas tolerantes a fallos, esta tesis se ha centrado en un método de validación experimental: las técnicas de inyección de fallos basadas en la simulación de modelos en VHDL. Las principales ventajas de este conjunto de técnicas son que se pueden aplicar en la fase de diseño del sistema y que permiten acceder a cualquier elemento del modelo del sistema. Por contra, presentan el inconveniente de que, sobre todo en modelos de sistemas complejos, la inyección de los fallos supone un elevado coste temporal. Sin embargo, sus importantes ventajas las hacen lo suficientemente atractivas como para ser utilizadas al menos como técnica complementaria de otras más utilizadas por su bajo coste y sencillez de implementación, como SWIFI (*software implemented fault injection*).

Un aspecto muy importante de las técnicas de inyección de fallos mediante simulación es la representatividad de los modelos de los fallos que se inyectan. En este sentido, se ha propuesto un amplio conjunto de modelos de fallos en los niveles lógico y RT, relacionados con los mecanismos físicos de los fallos en las nuevas tecnologías submicrónicas.

Otro asunto relacionado con las técnicas de inyección en las que se centra esta tesis es su automatización e integración en una herramienta de inyección de fallos. De las diferentes variantes que existen, se ha enfatizado en tres: las basadas en órdenes del simulador, perturbadores y mutantes. Tradicionalmente, estas dos últimas han tenido serios problemas de automatización. En esta tesis se han propuesto sendos métodos de implementación y automatización.

Con las premisas anteriores, se ha desarrollado VFIT (*VHDL-based Fault Injection Tool*), una herramienta de inyección de fallos sobre modelos en VHDL para PC. VFIT es independiente del modelo, y capaz de implementar las tres técnicas elegidas, con un amplio abanico de modelos de fallos.

La herramienta de inyección de fallos desarrollada se ha aplicado sobre diversos modelos en VHDL de sistemas, obteniendo resultados importantes en los campos de representatividad de los fallos, identificación de debilidades en los sistemas bajo estudio y su posterior corrección, así como de validación general de estos sistemas mediante predicción de fallos.

Abstract

When designing computer systems (and particularly those in which a wrong operation can provoke loss of human lives, economical damage, suspension of primary services, etc.), to get that they work properly as long as possible and with a high accuracy level, becomes an essential priority.

The systems that manage critical services have special mechanisms that provide some degree of immunity to the occurrence of failures able to cause a suspension or degradation of the service given. Thus, they are called Fault Tolerant Systems, or FTSs.

The concept of Dependability is defined as a set of functions (or attributes) that allow quantifying the quality of the service given with regard to the failures occurred, and consequently, the degree of confidence that the user can rely in the system. When developing a fault tolerant system, it is necessary to validate it, that is to say, to quantify its Dependability parameters.

Among the various existing methods and techniques used to validate fault tolerant systems, this Ph.D. Thesis has focused a method of experimental validation: the VHDL simulation-based fault injection techniques. The main advantages of this set of techniques are that they can be applied in design phase of the system, and that they allow accessing any elements of the system model. Instead, they have the drawback that mainly in models of complex systems, the injection of the faults implies a high temporal cost. Nevertheless, their important advantages make them attractive enough to be used at least as a technique complementary of other more popular because of their low cost and implementation simplicity, like SWIFI (*software implemented fault injection*).

A very important aspect of VHDL simulation-based fault injection techniques is the representativeness of the fault models injected. In this way, a wide fault model set for the logic and RT levels has been proposed. These models are related to the physical fault mechanisms in the new deep submicron technologies.

Another subject related to the injection techniques in which this Ph.D. Thesis focuses is their automation and integration in a fault injection tool. Three of the different implementations that exist have been selected: those based on the use of simulator commands, saboteurs and mutants. Traditionally, the two latter have had serious automation problems. In this Ph.D. Thesis, a method of implementation and automation for each technique has been proposed.

With the basis of previous premises, VFIT (*VHDL-based Fault Injection Tool*), a VHDL simulation-based fault injection tool to run under PC has been developed. VFIT is model-independent, and it is able to apply the three fault injection techniques selected, using a wide range of fault models.

The fault injection tool developed has been applied to several VHDL models, getting important results in the field of fault representativeness, in the detection of weak points in the fault tolerance mechanisms and their subsequent correction, and in the general validation of FTSs by means of fault prediction.

1 Presentación

1.1 Fundamentos y motivación

Los sistemas informáticos ocupan un papel cada vez más importante en nuestra vida cotidiana. Muchos de los servicios y comodidades que disfrutamos están controlados por sistemas informáticos. Desde los sencillos computadores personales o pequeños electrodomésticos que se usan a diario, hasta la gestión de los sistemas de transporte (control de tráfico rodado, ferroviario, aéreo y espacial), los sistemas bancarios, los sistemas militares, o los complejos sistemas industriales o de instalaciones civiles (plantas de energía, presas, etc.), todo está controlado por sistemas informáticos. Por esta razón, se establece como una prioridad esencial conseguir que los sistemas informáticos que hacen nuestra vida más confortable lo hagan durante el mayor tiempo posible y con un elevado nivel de eficacia.

De entre todos ellos, aquellos sistemas en los que, por las características del servicio que prestan, un mal funcionamiento puede provocar problemas serios (en el sentido de pérdida de vidas humanas, de fuerte perjuicio económico, de suspensión de servicios primordiales, etc.), requieren de una dedicación especial por parte de los diseñadores.

Estos sistemas disponen de unos mecanismos especiales que les proporcionan una cierta inmunidad a la ocurrencia de averías que puedan causar un cese o deterioro del servicio prestado. Por ello, a estos sistemas se les denomina **Sistemas Tolerantes a Fallos**, o STF.

Con el fin de cuantificar y/o incrementar la confianza que el usuario puede tener en un sistema, se debe llevar a cabo una evaluación de éste, sobre todo si está destinado a realizar una tarea crítica. Se denomina **Confiabilidad** al conjunto de funciones (llamadas atributos) que permiten cuantificar la calidad del servicio prestado (en cuanto a averías producidas[1]), y el grado de confianza que el usuario puede depositar en el sistema.

Los atributos que permiten cuantificar la Confiabilidad se basan en los diferentes puntos de vista bajo los que se puede determinar la calidad del servicio. Por ejemplo, si se comparan los sistemas de control de una sonda espacial y de una central nuclear, su calidad de servicio se mide desde dos criteriors muy diferentes. Del sistema computador que gobierna la sonda espacial interesa que funcione ininterrumpidamente el mayor tiempo posible, ya que su reparación es muy difícil (y costosa) si no imposible. Este atributo se denomina **Fiabilidad**. Por el contrario, al sistema computador de la central nuclear se le exigirá prioritariamente que no tenga averías catastróficas. Este atributo se llama **Seguridad-Inocuidad.**

La valoración de la calidad del servicio que presta un sistema (**validación**) se puede realizar de dos formas: **teórica** y **experimental**. La validación teórica se lleva a cabo resolviendo un modelo teórico del sistema, generalmente expresado mediante cadenas de Markov o redes de Petri estocásticas. La validación experimental se basa en observar el comportamiento ante fallos del sistema real o de un modelo de simulación del mismo. El principal problema de la la validación teórica es que, para resolver los modelos, se requieren ciertos parámetros del sistema difícilmente estimables de forma teórica, como los coeficientes de cobertura en la detec-

[1] En general, la calidad del servicio prestado por un sistema no tiene por qué tener en cuenta las averías (de ahí la matización). Sin embargo, en este trabajo de tesis se hablará simplemente de "calidad de servicio", si bien hay que aclarar que en realidad este concepto se trata desde el punto de vista de la existencia o no de averías.

ción y/o en la recuperación de errores, los tiempos de latencia en la propagación de los errores, y en la detección y/o recuperación por parte de los mecanismos de tolerancia a fallos.

Existen dos métodos básicos para realizar la validación experimental de un sistema: la **observación del sistema** durante su fase operativa, y la **inyección de fallos**. El primer método tiene como principal inconveniente la baja ocurrencia de fallos en sistemas tolerantes a fallos, por lo que para obtener datos estadísticamente representativos es necesario un tiempo de observación muy elevado. Por contra, la inyección de fallos consiste en provocar fallos de forma deliberada en el sistema bajo estudio (bien sobre un modelo de simulación o sobre un prototipo). Además, la inyección de fallos también sirve como apoyo a la validación teórica, aportando al modelo teórico los parámetros mencionados anteriormente.

La inyección de fallos, a su vez, se puede llevar a cabo mediante diferentes técnicas, que se pueden clasificar bajo diversos puntos de vista: la clase de sistema (prototipo o modelo de simulación), la forma en que se lleva a cabo (mediante un dispositivo físico o un programa), etc. Las técnicas de inyección son, a grandes rasgos, tres: física (o HWIFI, del *inglés hardware implemented fault injection*), implementada por *software* (o SWIFI, del *inglés software implemented fault injection*), y 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. Tras la realización de sendas herramientas basadas en HWIFI [Gil 1992, Gil *et al.* 1997a, Martínez *et al.* 1999] y SWIFI [Campelo 1999], el Grupo se planteó la realización de otra herramienta basada en simulación.

Los modelos de simulación pueden ser analíticos (basados en redes de Petri estocásticas, cadenas de Markov, modelos de colas, etc.), esquemas gráficos, o estar especificados mediante otros métodos de descripción.

En la actualidad, para el diseño y simulación de sistemas digitales se utilizan unos lenguajes especiales, llamados lenguajes de descripción de *hardware* (*hardware description languages*, o HDL). Uno de los más extendidos es 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. Se eligió VHDL porque, además, su semántica ofrece además propiedades que lo hacen muy adecuado para inyectar fallos.

La inyección de fallos mediante simulación de modelos en VHDL tiene otras ventajas, como cierta variedad de maneras en que se puede llevar a cabo, la posibilidad de inyectar un buen número de modelos de fallos, y sobre todo, un acceso absoluto a todos los elementos del modelo (tanto en lectura como en escritura), que le proporciona elevadas accesibilidad y controlabilidad.

Un aspecto muy importante de las técnicas de inyección de fallos mediante simulación en general es el modelado de los fallos que se pueden inyectar. Estos **modelos** dependen del nivel de abstracción con el que se haya realizado el modelo. En cualquier caso, lo que sí es determinante es que sean **representativos** de lo que sucede en realidad en los niveles más bajos de los sistemas: el silicio con el que se forman los transistores, o lo que es lo mismo, el nivel físico. Por este motivo, se ha considerado el estudio de la representatividad de los modelos de fallos como una de las principales tareas de este trabajo de tesis.

1.2 Objetivos

El objetivo principal del presente trabajo de tesis es estudiar los diferentes aspectos que hay que tener en consideración en la técnica de inyección de fallos mediante simulación de modelos en VHDL[2]: la representatividad de los modelos de fallos y las diferentes maneras (o subtécnicas) en que se puede llevar a cabo la inyección de fallos sobre modelos en VHDL.

En lo que se refiere al estudio de la representatividad de los modelos de fallos, se estudiará la literatura más actual para deducir cuáles van a ser los modelos de fallos más característicos de las nuevas tecnologías submicrónicas de circuitos integrados. De este estudio se deducirá un conjunto de modelos de fallos para los niveles de abstracción lógico y de transferencia de registro. La razón por la que se eligen estos niveles de abstracción y no otros es su proximidad al nivel físico, lo que hace que los modelos sean más eficientes.

En cuanto a las subtécnicas de inyección de fallos sobre modelos en VHDL, se seleccionará un subconjunto: órdenes del simulador, perturbadores y mutantes. A la vista de la problemática existente en algunas de estás técnicas (perturbadores y mutantes), además, se propondrán nuevas alternativas de implementación que se pueden aplicar de forma automática a cualquier modelo.

A causa de la consecución del primer objetivo, se plantea un segundo: la implementación de una herramienta de inyección que integre los modelos de fallos y métodos de inyección desarrollados. La herramienta que se plantea realizar debe tener las siguientes características:

- Ser una aplicación autónoma.
- Ser utilizable en una plataforma de uso común: un PC.
- Ser independiente del modelo.
- Conseguir un elevado nivel de automatización.
- Permitir la inyección de un amplio conjunto de modelos de fallos.
- Utilizar un variado rango de técnicas de inyección.

Con todas estas premisas, se abordará la realización de VFIT (*VHDL-based Fault Injection Tool*), una herramienta que permita inyectar fallos mediante las tres subtécnicas indicadas, y que incorpore los modelos de fallos deducidos.

El tercer objetivo que se plantea en el desarrollo de esta tesis es la aplicación de la herramienta desarrollada para inyectar fallos sobre diversos modelos de sistemas, y llevar a cabo diferentes estudios.

Estos estudios consisten tanto en el análisis del síndrome de errores de sistemas no tolerantes a fallos, como en la validación de sistemas tolerantes a fallos. Los resultados que se pretende obtener de la aplicación de VFIT dependen del tipo de estudio. En el análisis del síndrome de error, consistirán principalmente en la clasificación de los errores propagados y en el cálculo de sus latencias de propagación. En el caso de la validación, se calcularán las

[2] En este trabajo de tesis también se la referirá como *inyección de fallos sobre modelos en VHDL*.

coberturas de detección y recuperación de errores por parte de los mecanismos de tolerancia a fallos del sistema, y las latencias en la detección y recuperación de los errores.

1.3 Desarrollo

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

El capítulo 2 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.

En el capítulo 3 se describe con detalle el "estado del arte" tema de inyección de fallos, y se explican las diferentes técnicas existentes. Para todas ellas (con excepción de la inyección de fallos sobre modelos en VHDL, a las que se dedica un capítulo aparte), se describen las distintas variantes posibles, y se enumeran las herramientas más representativas.

La problemática del modelado de los fallos se recoge en el capítulo 4. En él se analizan los mecanismos de fallo que más importancia tienen (o se prevé que van a tener) en los circuitos integrados submicrónicos actuales. De este estudio se deduce un conjunto de modelos de fallos transitorios, permanentes e intermitentes, así como la tendencia en cuanto a sus tasas de fallo. Una de las conclusiones más destacables es el fuerte incremento de las tasas de los fallos transitorios e intermitentes.

En el capítulo 5 se completa el "estado del arte" referente a la inyección de fallos con las diferentes técnicas de inyección sobre modelos en VHDL. Al igual que en el capítulo 3, se describen las distintas variantes existentes y se enumeran las herramientas de inyección más relevantes, resaltando en cada una de ellas, como aspecto importante, los modelos de fallos que permiten aplicar. Sin embargo, en este capítulo, se incluyen además nuevos diseños para la implementación de algunas de las subtécnicas (perturbadores y mutantes).

El capítulo 6 está dedicado a VFIT, la herramienta de inyección de fallos desarrollada. Se indican sus capacidades y características. También se describe su estructura, tanto a nivel general como en detalle, definiendo las funciones de cada elemento de la herramienta. Para completar la especificación de sus características, se incluyen dos ejemplos prácticos de utilización de la herramienta. Uno consiste en la validación de un sistema tolerante a fallos desarrollado en el propio GSTF, y el otro, en el análisis del síndrome de error de un microcontrolador comercial: el PIC. En cada ejemplo se enumeran los parámetros necesarios para su realización, el modo en que se introducen en la herramienta y los resultados que se obtienen, tal como los genera la herramienta.

En el capítulo 7 se describen los resultados más interesantes obtenidos de la inyección de fallos con VFIT. Para ello, se describen los modelos de los sistemas sobre los que se ha aplicado, y los diferentes experimentos de inyección llevados a cabo. Para cada uno se especifican los objetivos buscados y los parámetros necesarios para realizar la inyección, y se muestran los resultados extraídos, en forma de tablas y gráficos, indicando los valores de latencias, coberturas, etc., calculados.

En el capítulo 8 se hace una recapitulación del trabajo presentado, y se extraen las conclusiones más interesantes a las se llega durante el desarrollo de todo el trabajo.

Por último, este trabajo se completa con un apéndice en el que se describen algunas funciones de distribución utilizadas para la generación aleatoria de valores, bien ya incorporadas en VFIT o que está previsto hacerlo en un futuro próximo.

2 Generalidades de los Sistemas Tolerantes a Fallos

En este capítulo se introducen y describen una serie de conceptos y términos utilizados en el campo de la tolerancia a fallos. Dichos conceptos se pueden interpretar de diferentes maneras, algunas de ellas incluso contradictorias. Estas divergencias en la interpretación surgen de las diversas traducciones al castellano que se realizan de los términos originales en inglés o francés. En este sentido, la terminología y definiciones que se aportan en este trabajo siguen las tendencias mostradas en [LIS 1996] y [Gil 1996], puntos de referencia en este campo de investigación.

2.1 Definiciones básicas

Confiabilidad[3] es la propiedad de un sistema informático que permite depositar una confianza justificada en el **servicio** que proporciona. El *servicio* proporcionado por un sistema es el comportamiento percibido por sus usuarios; un **usuario** es otro sistema (físico o humano) que interactúa con el primero.

En función de la aplicación particular a la que está destinado el sistema, se pueden considerar diferentes aspectos de la *Confiabilidad*. Esto significa que la Confiabilidad puede ser vista de acuerdo a diferentes, aunque complementarias, propiedades, lo que permite definir sus **atributos**:

- **Disponibilidad**[4]: Preparación para el servicio.
- **Fiabilidad**[5]: Continuidad del servicio.
- **Seguridad-Inocuidad**[6]: Ausencia de consecuencias catastróficas en el entorno.
- **Confidencialidad**[7]: Ausencia de revelaciones de información no autorizadas.
- **Integridad**[8]: Ausencia de alteraciones indebidas de información.
- **Mantenibilidad**[9]: Capacidad para someterse a reparaciones y evoluciones.

La asociación de la *Integridad* y la *Disponibilidad* respecto a acciones autorizadas, junto con la *Confidencialidad*, da lugar a la **Seguridad-Confidencialidad**[10].

Si se define la **función** del sistema como la tarea que **debe** llevar a cabo, y el **comportamiento** como el servicio proporcionado (es decir, lo que **hace** realmente), 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 hipotética de un error.

---

[3] En inglés, *Dependability*. Anteriormente se la conocía como Garantía de funcionamiento [Gil 1996].
[4] En inglés, *Availability*.
[5] En inglés, *Reliability*.
[6] En inglés, *Safety*.
[7] En inglés, *Confidentiality*.
[8] En inglés, *Integrity*.
[9] En inglés, *Maintainability*.
[10] En inglés, *Security*.

Resumiendo, 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**[11] (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.

La *Prevención* y la *Tolerancia a fallos* son métodos destinados a la **consecución** de una alta Confiabilidad; es decir, pretenden dar al sistema la capacidad de proporcionar el servicio que de él se espera.

Por contra, la *Eliminación* y la *Predicción de fallos* son métodos para la **validación** de sistemas; esto es, para cuantificar y justificar la confianza que se puede depositar en que el sistema sea capaz de proporcionar el servicio requerido.

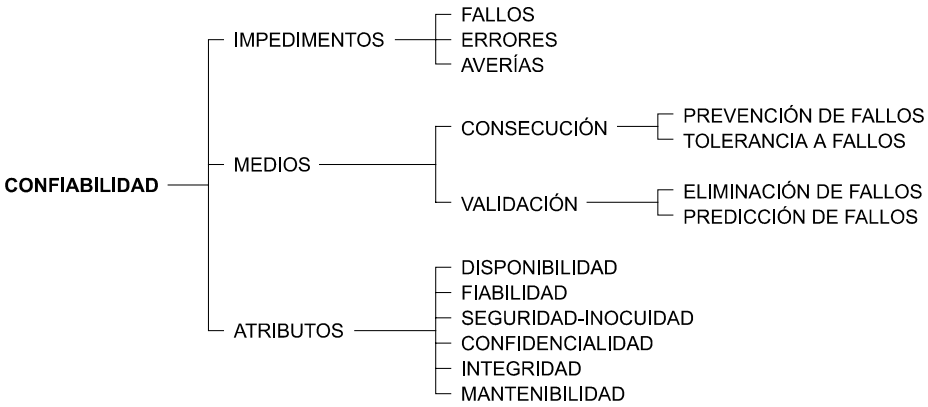

Figura 2.1: Árbol de la Confiabilidad.

Los conceptos presentados se pueden relacionar con la Confiabilidad como se muestra en la Figura 2.1, viendo la Confiabilidad como un conjunto compuesto por:

- Los **atributos** que hay que alcanzar. Estos permiten, por un lado, expresar las propiedades que se esperan de un sistema, y por otro, valorar la calidad del servicio proporcionado.
- Los **impedimentos** que se oponen a la consecución de los *atributos*. Son circunstancias no deseadas, si bien no inesperadas en principio, que causan o son el resultado de una falta de Confiabilidad. Dependiendo de cómo afecten al servicio, pueden incidir en la confianza del usuario en el sistema.

[11] En inglés, *Dependable systems*.

- Los **medios** para conseguir la Confiabilidad, que son métodos y técnicas para:
  1. Proporcionar al sistema la capacidad para entregar un servicio en el que se pueda confiar.
  2. Evaluar y cuantificar la confianza en esta capacidad.

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 profundiza un poco más en los elementos del árbol de la Confiabilidad, con el fin de poder justificar la necesidad de evaluar (validar) un sistema para así poder cuantificar la confianza que sus usuarios pueden tener en el servicio proporcionado.

2.2 Atributos de la Confiabilidad

Atendiendo a las definiciones dadas en el apartado anterior, se puede observar que:

- La *Disponibilidad* y la *Fiabilidad* están relacionadas con la capacidad de evitar las averías.
- La *Seguridad-Inocuidad* da idea de la capacidad de evitar una clase específica de averías (las catastróficas), y la *Seguridad-Confidencialidad* es la capacidad de prevenir lo que puede ser visto como una clase específica de fallos (el acceso y/o manipulación no autorizada de información).

Es decir, la *Fiabilidad* y la *Disponibilidad* están mucho más próximas entre sí que respecto a la *Seguridad-Inocuidad* y la *Seguridad-Confidencialidad*. Por este motivo, la *Fiabilidad* y la *Disponibilidad* pueden ser consideradas conjuntamente, pudiendo ser definidas globalmente como la minimización de los cortes en servicio. Sin embargo, este comentario no debería conducir a pensar que la *Fiabilidad* y la *Disponibilidad* no dependen del entorno del sistema. De hecho, desde hace mucho tiempo se reconoce que la *Fiabilidad*/*Disponibilidad* de un sistema informático está sumamente correlacionada con su perfil de utilización, sea por las averías debidas a fallos físicos, o por las debidas a fallos de diseño.

El hecho de que un sistema tenga las propiedades que han posibilitado la definición de los atributos de la Confiabilidad debe ser interpretado de forma relativa (probabilística), y no absoluta (determinista), ya que debido a la presencia inevitable de los fallos, los sistemas no son nunca totalmente disponibles, fiables, seguros-inocuos ni seguros-confidenciales.

Los atributos de la Confiabilidad se pueden definir de manera más formal. Por ejemplo, en [Gil 1992] se definen algunos de ellos:

- **Fiabilidad**: La Fiabilidad de un sistema en un instante t, R(t), es la probabilidad condicional de que el sistema, estando funcionando correctamente (en estado de servicio adecuado) en un instante t_0, continúe funcionando sin interrupciones durante el intervalo $[t_0,t]$, con $t > t_0$. Evidentemente, la Fiabilidad debe utilizarse en el estudio de sistemas sin reparación.
- **Disponibilidad**: La Disponibilidad de un sistema en un instante t, A(t), es la probabilidad de que el sistema funcione correctamente en el instante t. Nótese que no se trata de una probabilidad condicional, y que el sistema ha podido fallar antes del tiempo de estudio t. La Disponibilidad se utilizará, pues, en el estudio de sistemas con reparación.

- **Mantenibilidad, M(t)**: Es una medida del tiempo de reparación o restauración del sistema desde la última avería ocurrida. Su definición es similar a la de la *Fiabilidad*, pero referida al tiempo transcurrido entre el estado de servicio inadecuado y el de servicio adecuado.
- **Seguridad-Inocuidad**: La Seguridad-Inocuidad de un sistema en un instante t, 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 $t > t_0$. En los sistemas en los que se estudia la Seguridad-Inocuidad no tiene mucho sentido su estudio bajo el punto de vista de la reparación, ya que en ellos tiene mayor interés la importancia de las averías producidas que su posible reparación. Este parámetro se utiliza en el estudio de los sistemas críticos (sistemas que ponen en juego vidas humanas).

2.3 Impedimentos de la Confiabilidad

En este apartado se examinan los conceptos de **avería**, **error** y **fallo**, así como sus mecanismos de manifestación, es decir, la patología de los fallos.

2.3.1 Averías

La definición de *avería* realizada en el apartado 2.1 considera el incumplimiento de la función del sistema, no el de su especificación. Si se identificara como avería únicamente un comportamiento inaceptable debido a la no conformidad con la especificación, podría darse el caso de que un resultado cumpliera con la especificación y sin embargo fuera inaceptable para los usuarios del sistema, dejando al descubierto un fallo de especificación.

Generalmente, un sistema no se avería siempre de la misma manera. Las formas en que un sistema puede averiarse se denominan **modos de avería**, que 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:

a) **Salidas congeladas**. Se entrega un servicio de valor constante, que puede variar según la aplicación, siendo por ejemplo el último valor correcto, algún valor predeterminado, etc.

b) **Silencio**. Por ejemplo, no enviar ningún mensaje en un sistema distribuido.

Un sistema cuyas averías son solamente *con parada* se denomina, de manera general, **sistema con parada tras avería**[12]. Las situaciones de *salidas congeladas* o en *silencio* dan lugar, respectivamente, a **sistemas pasivos tras avería**[13] y a **sistemas en silencio tras avería**[14].

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; a menudo se les denomina **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.** Éste es el resultado de la ordenación de los *modos de avería* según diferentes niveles de gravedad, a los que van asociados generalmente las máximas probabilidades de ocurrencia.

El número, la denominación y la definición de los niveles de gravedad, así como las probabilidades de ocurrencia admisibles son, en gran parte, dependientes de las aplicaciones. Sin embargo, pueden definirse dos niveles extremos, de acuerdo con la relación entre el beneficio proporcionado por el servicio entregado en ausencia de avería y las consecuencias de las averías:

- **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 2.2 resume las anteriores clasificaciones de las averías.

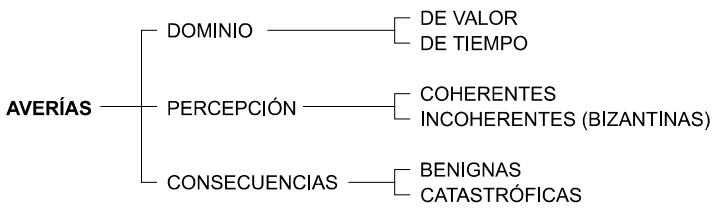

Figura 2.2: Clases de averías.

Un sistema donde todas las averías son (en una medida aceptable) *benignas* se denomina **sistema seguro tras avería** [15].

La noción de gravedad de las averías permite definir el concepto de **criticidad**: la *criticidad* de un sistema es la mayor gravedad de sus posibles modos de avería. La relación entre los

[12] En inglés *Fail-stop*.
[13] En inglés *Fail-passive*.
[14] En inglés *Fail-silent*.
[15] En inglés *Fail-safe*.

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.

2.3.2 Errores

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

1. De la composición del sistema, y en particular, de la naturaleza de la redundancia existente:
   - **Intencionada** o extrínseca (introducida para tolerar los fallos), 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*. En la práctica, a veces es más que difícil (por no decir imposible) construir un sistema sin alguna forma de redundancia.
2. De la actividad del sistema: Según esta actividad, puede suceder que no se utilice la parte que presenta el error, 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. Por ejemplo, en un sistema de transmisión, las averías dependen de la tasa de errores que el usuario considere como aceptable.

2.3.3 Fallos

Los fallos y las fuentes de fallo son sumamente diversos. Los fallos se pueden clasificar de acuerdo a cinco puntos de vista principales, que son:

- Sus **causas fenomenológicas**.
- Su **naturaleza**.
- La **fase de vida** del sistema en el que ocurren.
- Su situación respecto a las **fronteras** del sistema.
- Su **persistencia** (modo de manifestarse en el tiempo).

Las *causas fenomenológicas* distinguen entre **fallos físicos**, debidos a fenómenos físicos adversos y **fallos humanos**, que resultan de imperfecciones humanas.

Por la *naturaleza* de los fallos se diferencia entre **fallos accidentales**, que aparecen o son creados de manera fortuita, y **fallos intencionados**, que son creados deliberadamente, con o sin intención **maligna**.

En función de la *fase de vida* del sistema se puede distinguir entre **fallos de desarrollo**, resultantes de las imperfecciones originadas en el desarrollo del sistema (de la especificación de las necesidades a la implementación), durante las modificaciones posteriores, o bien durante el establecimiento de los procedimientos de explotación y mantenimiento del sistema, y **fallos de operación**, que aparecen durante la explotación del sistema.

Las *fronteras* del sistema clasifican los fallos en **internos**, que son aquellas partes del estado del sistema que, una vez utilizadas por la actividad computacional, producirán un error, y **externos**, que son el resultado de interferencias o de la interacción con su entorno físico (perturbaciones electromagnéticas, radiación, temperatura, vibración, etc.) o humano.

La clasificación en función de la *persistencia* temporal de los fallos los divide en:

- **Permanentes**, cuya presencia no está ligada a condiciones puntuales, sean **internas** (dependientes de la actividad computacional) o **externas** (dependientes del entorno).
- **Temporales**, cuya *presencia* está ligada a dichas condiciones y están, por tanto, presentes 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. Ejemplos de éstos son:
  - ⇨ Los fallos sensibles a patrones en las memorias semiconductoras y otros circuitos VLSI, cambios de parámetros en algún componente debidos a cambios de temperatura, retardos debidos a capacidades parásitas, etc.
  - ⇨ Los fallos debidos a situaciones ocurridas cuando el sistema alcanza un determinado nivel de carga (afectando tanto a la circuitería como a los programas), como problemas de retardos de propagación y de sincronización.
  - ⇨ Fallos debidos a diseños lógicos incorrectos, como los *hazards* dinámicos [McCluskey 1986].
  - ⇨ Fallos debidos al proceso de desgaste en los componentes electrónicos (metalizaciones, capa de óxido de los transistores, etc.).

La Figura 2.3 resume las diferentes clases de fallos que se han expuesto, según los diferentes puntos de vista considerados. Estas clases de fallos pueden denominarse **elementales**. En la práctica, un fallo determinado puede enclavarse en varias de estas clases, dependiendo del punto de vista con que se mire. Obviamente no todas las combinaciones son posibles, ya que algunas clases son excluyentes entre sí. En la Figura 2.4 se muestran las combinaciones válidas que se pueden producir, que dan lugar a cinco categorías, los **fallos físicos** propiamente dichos y cuatro categorías de **fallos humanos**:

- **Fallos de diseño**: incluye a 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.

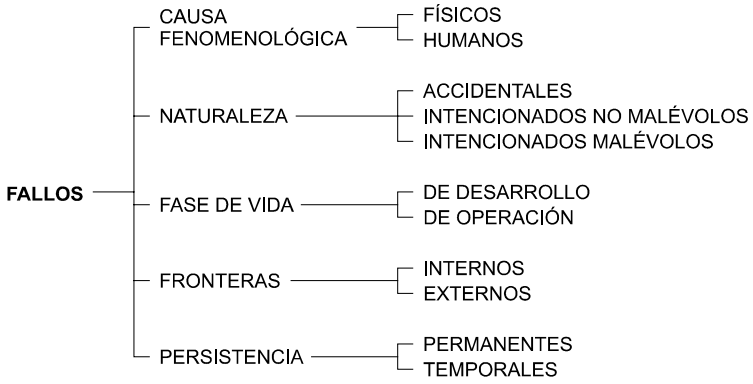

Figura 2.3: Clases de fallos elementales.

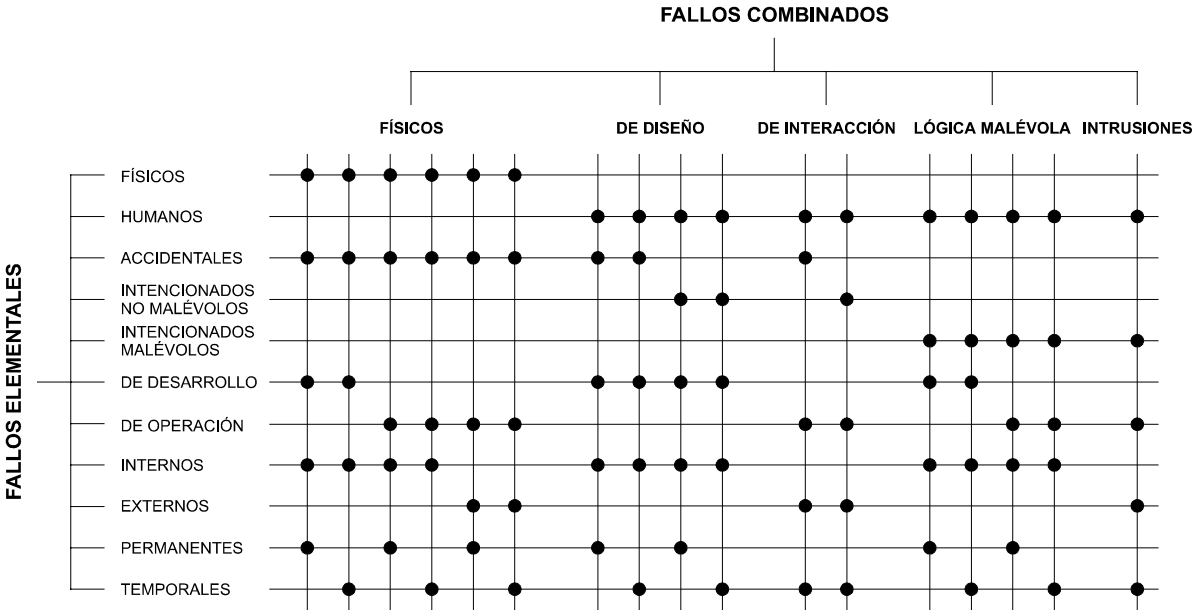

Figura 2.4: Clases de fallos combinados.

2.3.4 Patología de los fallos

Los mecanismos de creación y de manifestación de los fallos, errores y averías pueden resumirse de esta manera:

1. 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. La mayoría de los fallos internos oscilan entre sus estados dormido y activo. Los fallos físicos únicamente pueden afectar directamente a los

componentes materiales, mientras que los fallos humanos pueden afectar a cualquier componente.

2. 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.
3. Una avería ocurre cuando un error atraviesa la frontera sistema-usuario y afecta al servicio entregado por el sistema. Una avería de un componente da lugar, visto desde el punto de vista de los otros componentes que interactúan con él, a un fallo del sistema que contiene al componente, y los modos de avería del componente que se ha averiado dan lugar a diferentes tipos de fallo para los componentes que interactúan con él.

Estos mecanismos permiten completar la siguiente "cadena fundamental":

Fallo ➨ Error ➨ Avería ➨ Fallo ➨ ...

Las flechas de esta cadena expresan la relación de causalidad entre fallos, errores y averías. No obstante, no deben interpretarse de forma estricta, ya que mediante propagación pueden generarse varios errores antes de que ocurra una avería, y un error puede provocar un fallo sin que se haya observado una avería, ya que una avería es un evento que ocurre en la interfaz entre dos componentes.

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**.

Con frecuencia se encuentran situaciones donde están implicados múltiples fallos y/o averías. Tener en cuenta estos casos lleva a distinguir entre **fallos independientes**, atribuidos a causas diferentes, y **fallos conexos**, atribuidos a una causa común. Los *fallos conexos* se manifiestan con **errores similares**, mientras que los *fallos independientes* causan normalmente **errores distintos**, aunque puede suceder que a veces produzcan *errores similares*. Los errores similares causan **averías de modo común**. La generalización de la noción de errores similares da lugar al concepto de **errores coincidentes**, por ejemplo los errores que aparecen sobre la misma entrada. La relación temporal entre las averías múltiples lleva a distinguir entre **averías simultáneas** y **secuenciales**, según si ocurren en la misma ventana de tiempo predefinida o no.

2.4 Medios para alcanzar la Confiabilidad

En este apartado se analizan la *Tolerancia*, la *Eliminación* y la *Predicción de fallos*. La *Prevención de fallos* no se trata, 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. El apartado finaliza con una discusión acerca de la relación entre los medios para alcanzar la Confiabilidad.

2.4.1 Tolerancia a fallos

La Tolerancia a fallos se lleva a cabo mediante el procesamiento de los errores y el tratamiento de los fallos. El procesamiento de los errores está destinado a eliminarlos del estado computacional, a ser posible antes de que ocurra una avería. El tratamiento de los fallos está destinado a prevenir que se activen fallos de nuevo.

El procesamiento de los errores se puede llevar a cabo 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**[16], 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.

La sobrecarga temporal (en tiempo de ejecución) necesaria para el procesamiento de los errores puede ser radicalmente diferente en función de la técnica de recuperación de errores adoptada. En la recuperación hacia adelante o hacia atrás, la sobrecarga temporal es más importante cuando ocurre un error que en ausencia del mismo. Especialmente en la recuperación hacia atrás, este tiempo es consumido en el establecimiento de puntos de recuperación, es decir, en preparar al sistema para el procesamiento de los errores. Por otra parte, en la compensación de errores, la sobrecarga temporal es la misma, o prácticamente la misma, en presencia o ausencia de errores. Además, la duración de la compensación de un error es mucho menor que la de una recuperación de errores hacia adelante o hacia atrás, debido a la mayor cantidad de redundancia.

[16] En inglés, esta técnica recibe el nombre de *checkpointing*.

El primer paso en el tratamiento de los fallos es el **diagnóstico**, que consiste en la determinación de las causas de los errores, en términos de su localización y naturaleza. Posteriormente vienen las acciones destinadas a cumplir el objetivo principal del tratamiento de los fallos: impedir una nueva activación de los fallos, o sea, hacerlos pasivos. 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** (es decir, la modificación de la estructura del sistema), 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** o dulce; si se lleva a cabo la pasivación, el fallo se considera **duro** o sólido. A primera vista, las nociones de *fallo blando* y *duro* pueden parecer análogas a las de *fallo temporal* y *permanente*, introducidas anteriormente. Efectivamente, la tolerancia a fallos temporales no precisa del tratamiento de los fallos, ya que en este caso la recuperación del error deberá eliminar los efectos del fallo, que ha desaparecido por sí mismo (siempre que no se haya generado un fallo permanente en el proceso de propagación del temporal). Los conceptos de fallo blando y duro son útiles por los siguientes motivos:

- Distinguir un fallo temporal de uno permanente es una tarea compleja, ya que un fallo temporal desaparece después de un cierto intervalo de tiempo, normalmente antes de que se haya llevado a cabo el diagnóstico del fallo. Además, puede suceder que fallos de diferentes clases den lugar a errores similares. Por tanto, los conceptos de fallo blando y duro incorporan la subjetividad asociada a estas dificultades, incluyendo el hecho de que un fallo puede ser declarado como blando cuando su diagnóstico no ha tenido éxito.
- Por la capacidad de estos conceptos de incorporar sutilezas en los modos de acción de algunos fallos transitorios; por ejemplo, ¿se puede decir que un fallo dormido resultante de la acción de partículas alfa (debido a la ionización residual de los encapsulados de los circuitos), o de iones pesados del espacio sobre elementos de memoria es temporal?. Sin embargo, un fallo de este tipo es claramente blando.

2.4.2 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 de dos clases:

- Generales, que se aplican a una clase de sistema dado, y que son en consecuencia, (relativamente) independientes de las especificaciones. Ejemplos de condiciones generales

son la ausencia de bloqueos, o la conformidad con las reglas de diseño e implementación.

- 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:
  a) **Análisis estático**. Por ejemplo, inspecciones o ensayos, análisis del flujo de los datos, análisis de complejidad, verificaciones efectuadas por los compiladores, etc.
  b) **Pruebas de exactitud**. Por ejemplo, aserciones inductivas.
- 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 los **tests de verificación** (denominados comúnmente tests).

También es de especial importancia, respecto a lo que el sistema no debe hacer, verificar que el sistema no hace nada más que lo que está especificado. Este detalle está relacionado con la Seguridad-Inocuidad y la Seguridad-Confidencialidad.

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 el test**.

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*.

2.4.3 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, 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, que pueden por tanto verse como medidas de esta última.

Los métodos y herramientas para realizar la evaluación pueden ser específicos de cada modo (por ejemplo, análisis de modos y efectos de las averías para la *evaluación cualitativa*, o cadenas de Markov para la *cuantitativa*) o servir ambos para modos en conjunto (por ejemplo, los diagramas de bloques de la fiabilidad y los árboles de fallos).

La definición de las medidas de la Confiabilidad precisa, en primer lugar, de las nociones de servicio correcto e incorrecto. **Servicio correcto** es aquél donde el servicio entregado cumple con la función del sistema. **Servicio incorrecto** será aquél donde el servicio entregado no cumple con la función del sistema.

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 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**[17].

[17] En inglés, *Performability*.

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 **el test** (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 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. Estas últimas características pueden definirse 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 STF, 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 test, llamado en este caso **inyección de fallos**.

2.4.4 Dependencias entre los medios para alcanzar la Confiabilidad

En las definiciones de Prevención, Tolerancia, Eliminación y Predicción de fallos del apartado 2.1, se utiliza la palabra “cómo” para especificar los objetivos que cada mecanismo pretende alcanzar. En la realidad, todos estos objetivos no son completamente alcanzables, debido a la imperfección de la naturaleza humana, que interviene en todas actividades realizadas en aquellos mecanismos. Esto hace que la aplicación de uno de estos medios implique habitualmente la necesidad de aplicar otro u otros, por haberse introducido efectos colaterales en el sistema.

Por 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, por 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 deben 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 2.8.

2.5 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 [Arlat 1990].

La Prevención de fallos hace énfasis en la utilización 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 [Siewiorek y Swarz 1982]. 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 [Courtois *et al.* 1992].

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.

2.6 Confiabilidad y Validación

La Validación constituye uno de los principales problemas asociados al desarrollo y explotación de sistemas informáticos con funcionamiento garantizado. En efecto, 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?" [Laprie 1985].

La *Validación de la Cobertura* concierne principalmente a la validación del producto, es decir, 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 no obstante 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, como se indicó en el apartado 2.4.

La Eliminación de fallos consiste en reducir (mediante Verificación) la presencia de fallos y, en consecuencia, 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.

2.7 Tolerancia a fallos y Validación experimental

En los apartados anteriores se han enumerado diferentes aspectos de los Sistemas Tolerantes a Fallos (STF) 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 [Arlat 1990, Gil 1992]:

- 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 sus consecuencias (patología de fallos), sirvan de poco en diseños posteriores. Otros aspectos que hay que considerar son la cada vez mayor complejidad de los componentes, y 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 STF 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: 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 STF han evolucionado desde los llamados **macroscópicos** [Bouricius *et al.* 1969], 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** [Dugan y Trivedi 1989].

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 [Kanoun 1989].

Los modelos microscópicos están basados en la utilización de procesos estocásticos (procesos markovianos y/o semimarkovianos, redes de Petri estocásticas, etc.), e introducen técnicas de resolución analítica y/o de simulación. Se pueden aplicar al conjunto *hardware*/*software* del STF, 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 STF. 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).

2.8 Validación experimental e Inyección de fallos

La validación experimental puede llevarse a cabo de dos formas diferentes [Arlat 1990]:

- Mediante experiencias no controladas, observando el comportamiento en fase operativa de uno o varios ejemplares 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 observados y sus consecuencias son los que ocurren en el funcionamiento real del sistema. 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 STF[18]. 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 STF, por ser 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 STF, 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 STF. La técnica de *inyección de fallos* se define de la siguiente forma [Arlat 1990]:

Inyección de fallos es 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 STF 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 constituye un complemento indispensable de otros métodos de validación existentes (prueba formal, test simbólico, evaluación analítica, etc.). Hay complementariedad y no competencia [DBench 2001]. En efecto, con el fin de conseguir un máximo de confianza en el proceso de validación, es necesario aplicar conjuntamente varios métodos: 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 [Schwartz y Melliar-Smith 1983], EVE [Arlat *et al.* 1984], DELTA_4 [Powell 1988], IPDS (1 y 2) [Randell *et al.* 1995], HIDE [Majzik y Bondavalli 1998], GUARDS [Powell *et al.* 1999, Powell 2001] y DBench [DBench 2003]. 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 [FAST 2001]), mientras que la validación de la Confiabilidad del sistema se

[18] En sistemas tolerantes a fallos son comunes tasas de ocurrencia de un fallo cada 21 años ($5 \cdot 10^{-6}$ fallos/hora) [Laprie 1995].

realiza exclusivamente mediante técnicas de inyección (este es el caso del proyecto FIT [FIT 2002c]).

En la Figura 2.5 [Gil 1992] se puede apreciar el proceso de validación de un STF, tanto desde el punto de vista teórico como experimental, mediante inyección de fallos.

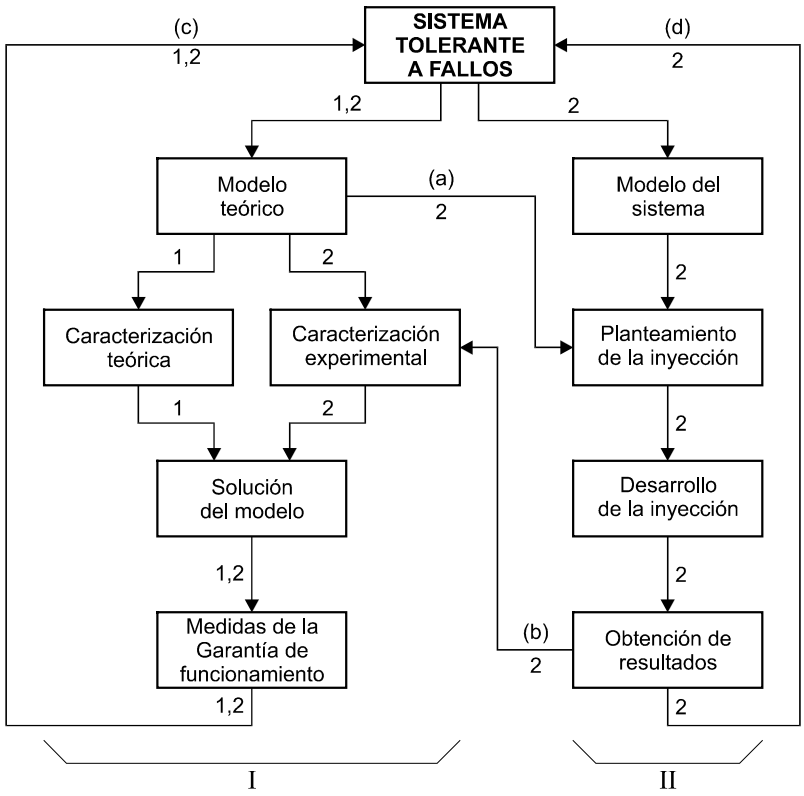

Figura 2.5: Diagrama de bloques del proceso de validación teórica y experimental mediante inyección de fallos. I: Predicción de fallos. II: Eliminación de fallos.

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, a 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 STF (realimentación (c)).

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).

2.9 Resumen y conclusiones

En este capítulo se ha definido el concepto de *Confiabilidad*, que es la propiedad de los sistemas informáticos que da idea de la confianza que los usuarios pueden depositar en el servicio que proporcionan.

Sin embargo, la Confiabilidad no es un concepto aislado. Por el contrario, engloba a 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*, *Integridad* y *Seguridad-Confidencialidad*.

Los impedimentos que 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.

En este sentido, se han clasificado las formas en que un sistema puede averiarse desde diferentes puntos 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 han clasificado los sistemas en función de su comportamiento ante las averías.

Los fallos, causa de los errores que ocasionan las averías, se han clasificado desde cinco puntos de vista (causa fenomenológica, naturaleza, fase de vida donde aparecen, situación respecto a las fronteras del sistema y persistencia), dando lugar a once tipos de fallos diferentes pero no excluyentes entre sí. Recombinando estos once fallos, se ha llegado a una nueva clasificación que da lugar a sólo cinco tipos de fallos: *físicos*, *de diseño*, *de interacción*, *lógica malévola* e *intrusiones*.

Los métodos que permiten alcanzar una elevada Confiabilidad, son cuatro: *Prevención de fallos*, *Tolerancia a fallos*, *Eliminación de fallos* y *Predicción de fallos*. La Prevención de fallos está íntimamente ligada con las técnicas de diseño. La Tolerancia a fallos 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 trata de reducir la presencia y la gra-

vedad de los fallos. Por su parte, la Predicción de fallos 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*.

Una de las misiones de la Validación es la medición de 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, lo que demuestra la fuerte relación entre la Validación y la 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 bien 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 *Inyección de fallos*, consistente en la introducción deliberada de fallos en un modelo o prototipo del sistema.

3 Técnicas de inyección de fallos

3.1 Introducción

Las técnicas de inyección de fallos se pueden clasificar desde diferentes puntos de vista [Hsueh *et al.* 1997, Yu 2001], como la **clase de sistema** sobre la que se aplica (el sistema real –o un prototipo–/un modelo), la **clase de fallos** que se inyectan (físicos[19]/*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 [Arlat 1992, Jenn 1994, Iyer 1995, Hsueh *et al.* 1997, Yu 2001]:

- Mediante **Simulación**. Se aplica sobre un modelo del sistema. Con esta técnica se pueden inyectar, dependiendo del nivel de representación del modelo, 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.
- Física, también denominada implementada mediante *hardware* (**HWIFI**, del inglés *hardware implemented fault injection*). Mediante esta técnica se pueden inyectar fallos físicos en el sistema real (o un prototipo), utilizando un mecanismo físico. El proceso de inyección se realiza actuando sobre la estructura material del "sistema", a través de medios físicos, para alterar su funcionamiento.
- Mediante ***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** [De Millo *et al.* 1987], que permiten inyectar fallos que representan fallos reales producidos en el *software*. Este tipo de técnicas se utilizan para validar el test de *software*, no para la Validación de la Confiabilidad, por lo que se salen del ámbito en el que se engloba el presente trabajo y no se considerarán.
  - ⇨ Emulación de fallos [Iyer 1995] (o **SWIFI**, del inglés *software implemented fault injection*). Mediante mecanismos *software* se puede emular fallos en el *hardware* y en el *software*. 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*.

En la Figura 3.1 se representan las diferentes técnicas descritas de manera esquematizada. En el presente trabajo se incidirá en las tres técnicas resaltadas con negrita en la figura: simulación, HWIFI y SWIFI.

Por otro lado, de entre las diferentes fases del ciclo de vida de un sistema (especificación, diseño, prototipado y operación)[20], sólo es posible evaluar el comportamiento del sistema en presencia de fallos en tres [Pradhan 1996]: diseño, prototipado y operación. De ellas, en las

[19] En el *hardware*.

[20] Existen otras nomenclaturas [Calvez 1993] en las que se consideran al menos cinco fases: especificación, diseño, implementación, producción y operación.

dos primeras se realiza mediante técnicas de inyección de fallos. Durante la fase de operación, lo más habitual es realizar la evaluación mediante técnicas basadas en la observación del sistema funcionando en su entorno de trabajo, obteniendo una serie de parámetros que dan idea de las tasas de avería, cuellos de botella, etc. [Iyer 1995]. Sin embargo, este tipo de evaluación está limitado a la ocurrencia, detección y recuperación de los errores reales, cuya probabilidad suele ser muy baja. Además, las condiciones de funcionamiento pueden variar ampliamente, afectando a la validez estadística de los resultados obtenidos.

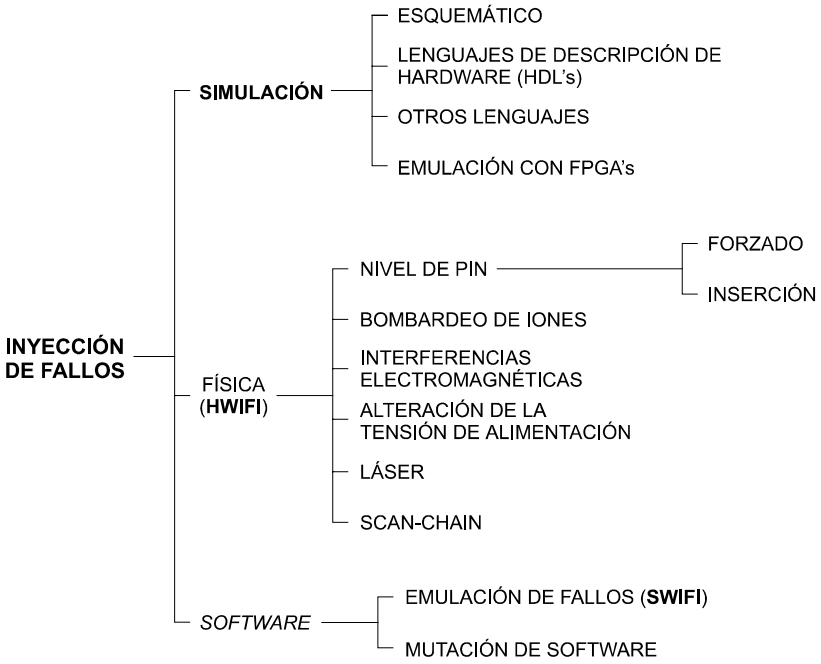

Figura 3.1: Clasificación general de las técnicas de inyección de fallos.

En cada una de las dos fases indicadas se pueden aplicar técnicas de inyección de diferente naturaleza. 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: HWIFI y/o SWIFI.

En la fase de diseño, para validar el modelo mediante simulación (incluyendo la inyección de fallos simulada) se utilizan herramientas CAD. La inyección de fallos comprueba en este caso 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, esto se complica a menudo debido a los cambios en el diseño y la tecnología.

En esta fase, 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 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.

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 (*Mean Time Between Failures*, MTBF) y la Disponibilidad (*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 3.2 a 3.4 se describen las características generales de las diferentes técnicas de inyección, así como algunos trabajos publicados y herramientas construidas para implementar automáticamente dichas técnicas en el proceso de Validación.

En el apartado 3.5 se realiza un análisis comparativo de las diferentes técnicas de inyección de fallos, atendiendo a una serie de criterios como la independencia del sistema sobre el que se aplica, los niveles de accesibilidad y controlabilidad sobre el sistema, su influencia sobre el sistema al que se aplica, el coste de la implementación, etc. La conclusión que se puede extraer de la comparación es que ninguna técnica es perfecta por sí misma, sino que entre ellas se complementan.

Además, con el auge en los últimos años 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. 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 utilizando una interfaz única. En el apartado 3.6 se describen algunas de estas herramientas más recientes.

3.2 Inyección de fallos mediante simulación

En la fase de diseño de un sistema, la simulación es un medio experimental importante para efectuar un primer análisis de las Prestaciones y la Confiabilidad. En efecto, cuanto más tarde se detecten y resuelvan los problemas de diseño que pueden aparecer, más costoso resulta no sólo en tiempo, sino también en dinero[21]. Comparándola con el modelado analítico, la simulación presenta la capacidad de modelar sistemas complejos con un alto grado de fiabilidad, sin restringirse a asunciones realizadas para conseguir que el modelo analítico sea matemáticamente tratable.

De las diferentes técnicas de este tipo mostradas en la Figura 3.1, la simulación de esquemas eléctricos y la de modelos en lenguajes de algún tipo (HDL, C, C++, ADA, etc.) son téc-

[21] 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 [Kilty 1995].

nicas basadas en una simulación por *software* de un modelo. Por el contrario, 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, *Programmable Logic Devices*) del tipo FPGA (*Field Programmable Gate Array*). Por ello, en el estudio de las distintas técnicas propuesto a continuación se tratará por separado la emulación con FPGA, por ser conceptualmente muy diferente de las otras si bien se parezcan en el hecho de que para llegar a realizar la implementación del sistema se haya tenido que utilizar un soporte lógico para modelar el sistema.

En cuanto a las técnicas basadas en simulación por *software*, la simulación para el análisis de la Confiabilidad implica la inyección de fallos sobre el sistema bajo estudio a diferentes niveles de abstracción. Por ejemplo, en [Pradhan 1996] se consideran tres niveles:

- Eléctrico. Se corresponde con los circuitos eléctricos, y los fallos que se inyectan 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 los bloques funcionales, los fallos inyectados son cambios en los registros de la CPU, *bit-flip* en la memoria, etc.

En [Jenn 1994] se especifican otros niveles: tecnológico, de circuito, lógico, de transferencia de registros (RT, del inglés *register transfer*) y no interpretado. [Siewiorek 1994] considera también tres niveles: de circuito, lógico y de sistema. En definitiva, 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 [Gil 1999].

La utilización de niveles de abstracción más elevados tiene la ventaja de que la complejidad de las simulaciones (tanto en espacio como en tiempo) disminuye. Así, por ejemplo, la simulación a nivel eléctrico no se puede usar de manera efectiva para estudiar sistemas VLSI complejos, y la simulación a 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. Por tanto, es evidente la necesidad de generar modelos lo más exactos posible. En el capítulo 4 se trata la problemática del modelado de fallos.

En cuanto a las características temporales de los fallos que se deben inyectar, hay que tener en cuenta que más del 80% de las averías de los sistemas computadores son debidas a fallos transitorios [Siewiorek 1994, Iyer y Rosseti 1986]. Estos fallos tienen diferentes causas físicas, como transitorios en la alimentación, diafonía[22] capacitiva o inductiva, o radiación cósmica.

Existen algunos problemas comunes que afectan a la inyección de fallos en todos los niveles de abstracción [Pradhan 1996]:

1. ¿Cuál es el modelo de fallos apropiado en el nivel de abstracción elegido? No hay una respuesta fácil a esta pregunta, de manera que los datos extraídos del funcionamiento real de los sistemas y la experiencia constituyen una valiosa guía.

[22] En inglés, *crosstalk*.

2. Para un modelo de fallo dado (por ejemplo, un *bit-flip* en la memoria) y un tipo de fallo (por ejemplo, uno transitorio), ¿dónde se debe inyectar el fallo? Una aproximación global consiste en elegir aleatoriamente un elemento del espacio de inyección (por ejemplo, todas las puertas de un chip, o todos los bits de memoria). Este esquema es fácil de implementar, pero muchos fallos pueden tener un impacto similar (por ejemplo, los fallos producidos en todos los bits de una ALU pueden tener el mismo efecto), y muchos lugares afectados pueden no ser sensibilizados. Otra aproximación es inyectar fallos en un conjunto reducido de lugares representativos con cargas (programas de prueba) selectivas, para realizar una validación dirigida del sistema.
3. El impacto de los fallos en la Confiabilidad del sistema depende de los programas de prueba (*workloads*). Por ello es importante analizar el sistema mientras ejecuta cargas representativas. Estas cargas pueden ser aplicaciones reales, *benchmarks* selectivos o programas sintéticos. Si el objetivo es investigar el impacto de los fallos en una tarea real, en la simulación se deben usar las aplicaciones reales. Si el objetivo es estudiar el impacto de los fallos con cargas generales, pueden utilizarse en la simulación varios *benchmarks* representativos. Si el objetivo es ejercitar todos los elementos y las unidades funcionales, se puede diseñar programas sintéticos para tal efecto. El problema de la carga complica los modelos de simulación, e incrementa el tiempo de simulación. Es esencial desarrollar técnicas para generar programas de prueba realistas, manteniendo al mismo tiempo tiempos de simulación razonables.
4. La "explosión" del tiempo de simulación. Esto ocurre en dos casos:
   a) Cuando se simulan muchos detalles, como en la inyección a nivel eléctrico.
   b) Cuando la probabilidad de los fallos es extremadamente pequeña, y se requiere una simulación larga para obtener resultados estadísticos significativos.

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. Dado que la herramienta presentada en el presente trabajo de tesis se enmarca en las técnicas de inyección mediante simulación sobre modelos en VHDL, este grupo de técnicas recibe un tratamiento más pormenorizado, y se describe en el capítulo 5. Por último, se describen las diferentes aproximaciones realizadas siguiendo la técnica de emulación de fallos con FPGA.

3.2.1 Nivel tecnológico

Los trabajos de [Choi e Iyer 1993] 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.

3.2.2 Nivel de transistor

Este nivel también se denomina de circuito, eléctrico o de dispositivo [Jenn 1994, Siewiorek 1994]. Hay diversas razones para simular la inyección de fallos en el nivel eléctrico:

1. 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.
2. Algunos estudios muestran que los modelos de fallos *stuck-at* y *bit-flip* no son siempre representativos de los fallos físicos [Galiay *et al.* 1980, Shen *et al.* 1985, DBench 2002].
3. Algunos circuitos tienen elementos analógicos y digitales, y no pueden ser caracterizados completamente por los modelos de fallos de tipo lógico. Se hace necesario por tanto el uso de simuladores de fallos que manejen fallos eléctricos transitorios y fallos físicos permanentes para los propósitos de test y validación de la Confiabilidad.

No obstante, 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.

En [Choi e Iyer 1992] se presenta la herramienta **FOCUS**, en la que se 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.

3.2.3 Nivel lógico

En ocasiones se le llama nivel de puerta (*gate-level*) [Siewiorek 1994]. La simulación de fallos a este nivel usa modelos lógicos abstractos tanto para los fallos como para las funciones de los circuitos. En contraste con la evaluación de los modelos físicos usados en el nivel eléctrico, la simulación a nivel lógico realiza operaciones binarias que representan el comportamiento de un dispositivo dado. Se obtienen salidas con valores discretos, y 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.

Las primeras aproximaciones se basan en la inyección de fallos *stuck-at* de tipo permanente. Así, en [Lomelino e Iyer 1986] se aplica sobre una descripción esquemática a base de puertas y biestables del procesador *bit-slice* AMD 2901, utilizando un simulador desarrollado por la NASA. En [Czeck y Siewiorek 1990] 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.

La herramienta descrita en [Cha *et al.* 1993, Cha *et al.* 1996] hace referencia a una serie de elementos que permiten una simulación eficaz de fallos a nivel lógico representativos de los fallos transitorios materiales (impacto de iones). Una primera fase consiste en obtener 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 almace-

nados en un registro del sistema, se efectúa una tercera y última fase de simulación de fallos paralela con "retardo-cero".

En [Choi e Iyer 1993] se propone una aproximación interesante para generar modelos de fallos reales a nivel lógico. Se denomina aproximación del diccionario de fallos (*fault-behavior dictionary approach*). Primero se simula a nivel eléctrico (por ejemplo con PSPICE) la zona donde se inyectan los fallos. 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 a 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.

3.2.4 Nivel de transferencia entre registros (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. En [Ohlsson *et al.* 1992] y [Rimén y Ohlsson 1993] 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 consiste en la inversión de un bit de un registro elegido aleatoriamente (*bit-flip*). En [Karlsson 1990] se comparan las técnicas basadas en la inyección de iones pesados e inyección simulada.

En [Yount y Siewiorek 1996] se propone la herramienta **ASPHALT**, que 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 (ver apartado 3.4) con la simulación a 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 efectuarse 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 a nivel lógico con fallos *stuck-at* transitorios, utilizando una aproximación similar a la del *diccionario de fallos* comentada anteriormente. En el estudio se indica que aproximadamente el 97% de los fallos del nivel lógico son cubiertos por los fallos RT.

3.2.5 Nivel de sistema

La simulación de fallos a este nivel se utiliza para estudiar sistemas computadores completos y redes de computadores, en lugar de sus componentes individuales. Estos estudios habitualmente consideran el *hardware*, el *software*, sus interacciones y la interdependencia entre los diversos componentes del sistema. Existen algunos problemas en el desarrollo de modelos de simulación a nivel de sistema:

a) Dificultad en establecer modelos de fallos.

b) Un espectro de componentes extenso y variado.

c) El esfuerzo y el tiempo requerido para desarrollar un modelo de simulación.

d) El impacto del *software* en la Confiabilidad.

e) La "explosión" del tiempo de simulación.

Dado que existen numerosas herramientas de modelado analítico, basadas principalmente en modelos de Markov y redes de Petri, surge la pregunta de cuál es le necesidad de las herramientas de simulación a nivel de sistema para el análisis de la Confiabilidad. ¿Qué información y prestaciones adicionales pueden aportar? La respuesta es que las herramientas de modelado analítico sólo usan modelos probabilísticos para representar el funcionamiento del sistema. Básicamente, el efecto de un fallo en el sistema está caracterizado por un conjunto de probabilidades y distribuciones. Por el contrario, la simulación a nivel de sistema no precisa de la caracterización del efecto de los fallos. Sólo es necesario conocer la tasa de llegada y los tipos de fallos. Por tanto, los resultados de la simulación pueden identificar los mecanismos de las averías, obtener su probabilidad y cuantificar el efecto de los fallos. Pueden usarse para extraer las características que se desea modelar, y ayudan a determinar y especificar la estructura y los parámetros de los modelos analíticos.

A continuación se describen algunas herramientas:

- **NEST** (*NEtwork Simulation Testbed*) es un entorno gráfico bajo UNIX cuyo objetivo es modelar, ejecutar y monitorizar sistemas distribuidos y protocolos [Dupuy *et al.* 1990]. Utilizando un conjunto de herramientas gráficas, el usuario puede desarrollar modelos de simulación y redes de comunicaciones. El modelo incluye las funciones de los nodos (por ejemplo, protocolos de comunicaciones) y el comportamiento de los enlaces (por ejemplo, pérdida de paquetes o retardos). 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. Estas simulaciones reconfigurables dinámicamente pueden utilizarse para estudiar el impacto de las averías en los nodos o enlaces, así como su recuperación.

- En **DEPEND** [Goswami e Iyer 1992, Goswami *et al.* 1997] se implementa una aproximación intermedia entre el modelado interpretado y el no interpretado. 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.

- La herramienta **REACT** [Clark y Pradhan 1992] permite la descripción de sistemas compuestos de un conjunto de procesadores y otro de bloques de memoria interconectados, con mecanismos de detección y tratamiento de errores (votador, código detector/corrector de error, etc.). El 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. Al 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.
- Los métodos propuestos en [Aylor *et al.* 1992] se aplican sobre un modelo no interpretado, basado en un formalismo próximo a las redes de Petri, y construido sobre el lenguaje VHDL. Esta aproximación permite 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** [Kumar *et al.* 1994, Ghosh *et al.* 1995a] utiliza este formalismo para efectuar la evaluación de los parámetros de sistemas distribuidos, analíticamente y mediante simulación.
- Otras herramientas y aportaciones:
  - ⇨ En [Güthoff y Sieh 1995] se presenta una herramienta híbrida que combina la técnica SWIFI con la de simulación para acelerar el proceso de inyección (de manera similar a ASPHALT), si bien en este caso el modelo se realiza en este caso a nivel de sistema.
  - ⇨ A nivel de sistema, el modelo de comportamiento consiste, típicamente, 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 testado 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 [Echtle y Chen 1991, Chen 1993].
  - ⇨ En [Maxion y Olszewski 1993] 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 [Jagannath y Rai 1995] 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 [Avresky *et al.* 1992] 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 [Goswami e Iyer 1993] se inyectan fallos en el espacio de memoria del sistema mientras se simula la ejecución de los programas.
  - ⇨ Por último, otro factor que hay que estudiar es la influencia de la carga de un sistema en la Confiabilidad. A partir de los trabajos [Meyer y Wei 1988, Bowen y Pradhan 1991, Czeck y Siewiorek 1992, Woodbury y Shin 1990, Iyer y Rosseti 1986] parece demostrada la influencia de la carga en la manifestación de los fallos que puedan ocurrir.

3.2.6 Emulación de fallos con FPGA

Esta técnica (también denominada *Emulación de fallos*[23] [Cheng *et al.* 1995] y *Emulación lógica* [Hwang *et al.* 1998]) se basa en el uso de FPGA para realizar prototipos preliminares del sistema en desarrollo. Para la especificación del modelo que se implementará en las FPGA se suelen utilizar lenguajes de descripción de *hardware* (VHDL o Verilog, principalmente), 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 [Leveugle 2001].

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 test.

En las primeras aproximaciones a esta técnica [Li y Wu 1995, Cheng *et al.* 1995, Burgun *et al.* 1996, Hong *et al.* 1996, Li *et al.* 1997, Cheng *et al.* 1999], el instrumento sobre el que se implementaba el modelo y realizaba la inyección de fallos era un emulador lógico compuesto por varias placas con FPGA en paralelo, así como del *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 (denominada inyección de fallos estática) consistía en detener el funcionamiento del sistema en un momento determinado y reconfigurar (modificar la programación) las FPGA. La reconfiguración podía ser de todas las FPGA o sólo de una parte, y además total (la FPGA entera) o parcial (sólo algunas partes). El principal inconveniente de esta implementación es la gran cantidad de tiempo perdido en reconfigurar las FPGA.

Como alternativa surgió la inyección dinámica, consistente en modificar el diseño original para hacerlo "inyectable" (en inglés *fault-injectable*). Para ello, por un lado se le añaden entradas adicionales de control que indicarán si se inyecta un fallo o no, y dónde. Por otro, hay que modificar el diseño para que pueda realizar tanto la función correcta como versiones erróneas de la misma, controlando la activación de cada "versión" con 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.

A medida que la densidad de integración ha ido aumentando, se ha incrementado la capacidad de procesamiento de las FPGA, y han aparecido en el mercado placas de prototipado[24] (o de desarrollo) directamente conectables a un computador "maestro" [Leveugle 2001]. Estas placas han supuesto un gran avance ya que, además de su bajo coste comparado con el de un sistema basado en emulador, han eliminado la necesidad del *software* emulador, lo que aumenta la capacidad de automatización de la técnica. Por contra, las placas de desarrollo presentan un nuevo tipo de problema: el número de pines de entrada/salida disponibles para implementar funciones es bajo. Esto es así porque este tipo de tarjetas suelen disponer de dispo-

[23] No confundir con SWIFI (ver apartado 3.1).

[24] Este tipo de placas pueden ser comericalizadas tanto por las propias empresas fabricantes de FPGA's [Lima *et al.* 2001] como por otras empresas (terceras partes) [Leveugle 2001].

sitivos accesorios (visualizadores, señales de reloj, interruptores, pulsadores, *led*, etc.) preconectados a la FPGA. Para solventar el problema, en [Leveugle 2001] se propone añadir al modelo sendos elementos adicionales para que hagan de interfaz de entrada y salida. A través de ellos se 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, pero su inserción se puede automatizar.

En cuanto al modo en que se realiza la inyección de los fallos, la técnica seguida en [Leveugle 2000, Leveugle 2001] se basa en la generación de *mutantes* a nivel RT (véase el apartado 5.4.2).

En [Lima *et al.* 2001], en cambio, se propone el uso de *perturbadores* (véase el apartado 5.4.1) a nivel RT, que manejados mediante una serie de señales de control permiten modificar el contenido de registros y palabras de memoria, inyectando fallos de tipo *bit-flip* mediante una máscara.

Existen otras dos aproximaciones similares que se basan en mutantes a nivel de puerta [Velazco *et al.* 2001, Civera *et al.* 2001a, Civera *et al.* 2001b]. Para ello, necesitan que el modelo sea estructural a nivel de puerta[25], 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. La diferencia entre ambas técnicas consiste en que en [Velazco *et al.* 2001] la circuitería de inyección es puramente combinacional, mientras que en [Civera *et al.* 2001a, Civera *et al.* 2001b] el mutante del biestable es más sofisticado, conteniendo además de la circuitería de inyección, un segundo biestable conectado es serie con los de 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 que se describirá en el apartado 3.3.2.3. Este método es mucho más potente que el anterior, ya que permite inyectar fallos múltiples. Una de las dos versiones de esta técnica se ha materializado en una herramienta denominada FIFA (*Fault Injection by means of FPGA*) [Civera *et al.* 2001b].

3.3 Inyección de fallos implementada mediante *hardware*

La inyección de fallos implementada mediante *hardware* (*hardware implemented fault injection*, HWIFI) es un método de introducir 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 y no se pueden obtener con otros métodos de inyección de fallos, como la latencia de los fallos en la CPU.

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 toma como referencia los límites de un circuito integrado, se pueden distinguir dos bloques diferenciados: inyección externa e inyección interna.

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

3.3.1 Inyección de fallos externa

La inyección de fallos a nivel de *pin* es una de las técnicas HWIFI más frecuente. Su objetivo es alterar los valores lógicos de distintas señales sobre las que se efectúa la inyección [Schmid *et al.* 1982, Schuette *et al.* 1986, Finelli 1987, Damm 1988, Arlat 1990, Gil 1992, Madeira *et al.* 1994, Gil *et al.* 1997a, Martínez *et al.* 1999].

Dentro de la idea de alterar el nivel de los *pines* se destacan a su vez, dos modalidades: **forzado**, en la que el nivel 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.

Otros investigadores, en lugar de alterar los niveles de uno o más *pines* realizan breves conmutaciones en la fuente de alimentación [Damm 1986, Gunneflo 1990, Miremadi *et al.* 1992]. Esta técnica consiste en provocar fluctuaciones de tensión en la entrada de alimentación. Es posible variar la duración de la perturbación, así como el salto de tensión producido.

Recurrir a interferencias electromagnéticas (EMI) es otra forma de inyectar fallos en el exterior de un circuito integrado [Reisinger y Steininger 1994, Karlsson *et al.* 1995].

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

3.3.1.1 Inyección a nivel de pin

Como ya se ha mencionado, 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.

En esta técnica, los fallos se inyectan directamente en los *pines* de los circuitos integrados que componen el sistema. Existe una variedad de fallos que se pueden introducir:

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

Por otra parte, la duración de los fallos se puede determinar para que se obtengan tanto fallos transitorios, intermitentes o permanentes.

Desde el punto de vista de la implementación hay, como se indicaba anteriormente, dos técnicas:

- **Forzado**: Los fallos se inyectan directamente sobre los *pines* del circuito integrado.
- **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.

El lugar de inyección en estas dos técnicas está limitado a los *pines* de los circuitos integrados, y por debajo del nivel de *pin* la inyección es impracticable. 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, por ejemplo **FTMP** [Lala 1983], **MESSALINE** [Arlat *et al.* 1990], **RIFLE** [Madeira *et al.* 1994] y **AFIT** [Gil 1992, Gil *et al.* 1997a, Martínez *et al.* 1999] (desarrollada por el Grupo de Sistemas Tolerantes a Fallos, GSTF, de la Universidad Politécnica de Valencia).

También existe una herramienta comercial: **DVT-100** [Proteus 1996, Stewart 1997], comercializada por Proteus Corporation. Se trata de una herramienta automatizada, dotada de sondas (de inyección y monitorización) robotizadas.

3.3.1.2 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 [Karlsson *et al.* 1995] 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 [Reisinger y Steininger 1994].

3.3.2 Inyección de fallos interna

Los métodos más usados se basan en la radiación de iones. También se han realizado algunos experimentos de inyección mediante láser, y otros que aprovechan los puertos de test (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.

3.3.2.1 Inyección de iones pesados

Este método se basa en causar interferencias en el interior de un circuito integrado utilizando radiación de iones pesados [Gunneflo *et al.* 1989, Karlsson *et al.* 1989, Gaisler 1997, Gaisler 2002]. 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 [Miremadi *et al.* 1992, Miremadi y Torin 1995, Karlsson *et al.* 1995].

Para usar esta técnica, el circuito integrado debe ser desprovisto de su encapsulado, y dispuesto en un entorno al vacío junto con el material radioactivo. 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.

Un ejemplo de herramienta que sigue esta técnica es **FIST** (*Fault Injection System for Study of Transient Fault Effects*), que utiliza una fuente de Cf-252. Empleando FIST, en los experimentos de [Gunneflo *et al.* 1989] y [Karlsson *et al.* 1989] se investigó la cobertura de los errores y las latencias de detección del microprocesador de 8 bits MC6809.

En [Gaisler 1997] y [Gaisler 2002] se muestran los resultados de validar sendos procesadores tolerantes a fallos (ERC32 y LEON-FT) utilizando esta técnica mediante inyección utilizando un acelerador de partículas.

La aplicación de este método presenta serias dificultades técnicas. Además, los iones pesados desencadenan fenómenos que pueden dañar el circuito (*latch-up* y excesiva disipación de calor, entre otros).

3.3.2.2 Inyección mediante láser

Los primeros trabajos efectuados sobre este tipo de inyección de fallos se han basado en la ruptura de ciertas conexiones internas de los circuitos integrados mediante un rayo láser, produciendo fallos permanentes [Velazco *et al.* 1990, Martinet 1992].

Más recientemente se ha realizado algún estudio inyectando fallos transitorios. En [Samson *et al.* 1997, Samson *et al.* 1998] 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 (*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, y 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 [Samson *et al.* 1998] se presentan algunos experimentos de test 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.

3.3.2.3 Inyección de fallos mediante Scan-Chain

Dentro de la inyección de fallos interna, últimamente se ha recurrido a los mecanismos de testado 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*) [Folkesson *et al.* 1998]. A través del TAP (*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 [Folkesson *et al.* 1998] se apuntaba que se podían inyectar fallos transitorios, y se esperaba poder inyectar fallos permanentes. La forma de inyectar se basa en establecer un punto de parada cuando se desea realizar una inyección y efectuar, cuando esto ocurra, la alteración de alguna unidad funcional del microprocesador. En [Folkesson *et al.* 1998] se puede ver 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.

3.4 Inyección de fallos implementada mediante *software*

La comprobación y evaluación de STF se ha convertido en algo cada vez más complicado, debido en gran parte a la creciente densidad de integración de los circuitos integrados. Muchas de las técnicas físicas descritas solamente pueden forzar un fallo en los límites de un circuito integrado, es decir, a nivel de *pin*. Cada vez es más frecuente la inclusión en el mismo encapsulado, y más en entornos basados en microcontroladores, de diversas unidades funcionales (por ejemplo, los SOC, *Systems On a Chip*). Es en este punto, por tanto, donde la inyección de fallos implementada mediante *software* (*software implemented fault injection*, 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. En este método de inyección no es necesaria ninguna instrumentación adicional, y el usuario puede elegir los puntos de inyección tanto en el *hardware* como en el *software* del sistema, siempre que sean accesibles por las 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* [Pradhan 1996]. 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*. Evidente-

mente, esta emulación es costosa en tiempo, introduciendo la monitorización y ejecución de muchas instrucciones adicionales.

A diferencia de las técnicas de inyección implementadas mediante *hardware*, en las que es difícil de orientar hacia áreas específicas del *software*, la inyección mediante SWIFI puede afectar a aplicaciones del usuario, al sistema operativo, o a ambos. 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. Si se trata del sistema operativo, el inyector de fallos debe insertarse en éste[26], porque es muy difícil añadir una capa adicional entre la máquina y el sistema operativo.

Aunque esta técnica es flexible, presenta algunas restricciones:

1. No se puede inyectar fallos en posiciones no accesibles por *software*.
2. Las rutinas de inyección pueden perturbar la ejecución de la carga del sistema, e incluso cambiar la estructura del programa original. Es necesario un diseño cuidadoso del entorno de inyección para mitigar los efectos de este problema.
3. La resolución temporal de la técnica es 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). Este problema puede resolverse utilizando un monitor *hardware*, es decir, empleando una aproximación híbrida [Young *et al.* 1992]. La aproximación híbrida 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 (limitada a puntos concretos) y aumenta el coste.
4. La generalización del uso de una herramienta SWIFI (independientemente del método empleado) a diferentes sistemas es difícil, dado que todos los métodos se basan en las características (*hardware* y *software*) del sistema al que se aplica.

En cuanto a los métodos que se siguen para realizar la inyección, éstos se pueden clasificar en dos tipos básicos [Folkesson 1999]:

- Previo a la ejecución, en inglés *pre-runtime fault injection*). Consiste en modificar la carga del sistema antes de su ejecución. A su vez, existen dos variantes:
  - ⇨ Modificación en tiempo de compilación. Consiste en modificar el código fuente del programa que se desea alterar, para que se comporte de manera diferente, introduciendo fallos en el sistema. 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.

  Este tipo de técnica tiene la ventaja de minimizar la intrusión en el sistema.
- En tiempo de ejecución. 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 "re-

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

al". 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, aquéllos accesibles por *software*).

Se han publicado numerosos resultados de aplicar esta técnica. Sin embargo, normalmente se trata de estudios y herramientas poco generales, realizados ex-profeso para un sistema particular. La razón es que, aunque la inyección de fallos mediante SWIFI es relativamente sencilla de realizar, la aplicación de los posibles métodos 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*) [Segall *et al.* 1988] se desarrolló 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. Para generar los fallos se corrompía 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 su incapacidad de inyectar fallos permanentes (únicamente inyectaba transitorios), y que los tiempos de latencia no eran obtenidos de una forma precisa.
- En la Universidad de Dortmund se desarrolló otro inyector por *software*, llamado **EFI** [Echtle y Leu 1992], 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. De este modo, tanto los mensajes entrantes como salientes eran alterados: modificados, reordenados, omitidos, etc. En el mismo centro se desarrolló otra herramienta, denominada **ProFI** (*Processor Fault Injector*) [Lovric y Echtle 1993], con el objetivo de detectar los errores en el *hardware*. Se basa en inyectar fallos a nivel de registros y código máquina del procesador. Estos fallos son permanentes.
- El entorno **FERRARI** (*Fault and ERRor Automatic Real-time Injector*) [Kanawati *et al.* 1992, Kanawati *et al.* 1995] se desarrolló en la Universidad de Texas. El objetivo de esta herramienta es la evaluación de sistemas complejos emulando los fallos físicos. Implementado sobre una estación SUN SPARC, la inyección de fallos se realiza corrompiendo el estado de ejecución de un programa mientras está en ejecución, de forma que el estado sería el mismo que si hubiera sido causado por un error interno. Para conseguir esto se basan en 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. Esta herramienta puede inyectar tanto fallos permanentes como transitorios.
- En la Universidad de Michigan se realizó la herramienta **SFI** (*Software Fault Injector*) [Rosenberg y Shin 1993], aplicada a sistemas distribuidos, que soportaba inyección a bajo nivel en nodos, y a alto nivel entre los nodos del sistema. Permitía inyectar fallos transitorios, intermitentes y permanentes, pudiendo especificar los parámetros temporales y los distintos tipos de fallos. Concretamente, era capaz de inyectar fallos tanto en el procesador como en la memoria y el sistema de comunicaciones. La idea básica para inyectar fallos en el procesador era cambiar algunas instrucciones generadas por el compilador (es decir, era del tipo *pre-runtime*). **DOCTOR** (*integrateD sOftware im-*

plemented fault injeCTiOn enviRonment) [Han *et al.* 1994], se desarrolló a partir de SFI, 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*) [Kao *et al.* 1993] se desarrolló en la Universidad de Illinois. El objetivo de esta herramienta era estudiar la propagación de los fallos en el núcleo del sistema operativo UNIX. Por ello, FINE era capaz de emular fallos físicos e inyectar fallos de programa en el núcleo del sistema. En particular, era capaz de emular fallos permanentes, transitorios e intermitentes en el procesador, la memoria y los *buses*.

  Tanto el inyector como la monitorización de los fallos se encuentran empotrados en el núcleo del sistema, con el objetivo de facilitar la inyección y analizar la propagación del error. Como en casos anteriores, 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 corta. 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 [Kao e Iyer 1994] 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. Además, es capaz de inyectar fallos en la CPU y en el *bus*.

- **FTAPE** (*Fault Tolerance And Performance Evaluator*) [Tsai e Iyer 1995a, Tsai e Iyer 1995b], se ha desarrollado en la Universidad de Illinois. Su objetivo principal es ser una herramienta para comparar STF. Entre sus características principales, se puede destacar que recurre a carga sintética para generar actividad, mayoritariamente en la CPU, la memoria o el sistema de entrada/salida. Se puede elegir dónde y cuándo inyectar. Como la carga es configurable, se crean en la máquina lo que denominan "condiciones de estrés". De esta forma, surge el concepto de inyección de fallos basada en el estrés del sistema (*stress-based injection*), que garantiza una mayor efectividad del fallo en función del sistema, o de su uso.

- **Xception** [Carreira *et al.* 1998] ha sido desarrollada en la Universidad de Coimbra. Esta es una de las herramientas que usa los avances en depuración y monitorización que existen en algunos procesadores modernos para inyectar fallos y analizar los resultados. Es capaz de inyectar fallos permanentes y transitorios sobre el procesador, la memoria y los *buses* del sistema. En este caso no hay que modificar el fichero ejecutable, ya que gracias a las capacidades internas de depuración de los procesadores, se puede lanzar una rutina de interrupción tras un determinado tiempo o evento. En este caso 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*) [Rodríguez 1998, Fabre *et al.* 1999] es una herramienta desarrollada por investigadores del LAAS-CNRS en Toulouse. El objetivo que les lleva a desarrollar esta herramienta es la validación de sistemas basados en *Microkernels* COTS. Para evaluar estos siste-

mas, MAFALDA es capaz de corromper los parámetros de invocación a las llamadas del sistema y, además, 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.

Para corromper las llamadas al sistema se altera uno de los bits de los parámetros de forma aleatoria. 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.

- **SOFI** (*SOftware Fault Injector*) [Campelo 1999] es un inyector de fallos por *software* desarrollado por el Grupo de Sistemas Tolerantes a Fallos de la Universidad Politécnica de Valencia. Esta herramienta realiza la inyección del fallo siguiendo la técnica *runtime*; es decir, efectúa dicha inyección durante la ejecución de la aplicación del sistema bajo análisis. Para ello, SOFI utiliza un pequeño agente de inyección, que se tiene que montar junto con la aplicación del sistema bajo prueba, encargado de recibir las órdenes de un supervisor, que es quien determina el instante de la inyección, determinado de forma aleatoria en el tiempo.

  En cuanto al fallo, SOFI realiza una inyección que intenta emular fallos en la ALU, los registros del procesador y la memoria. Estos fallos se pueden conseguir 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. En referencia al tipo de fallo, se realiza una negación de la posición inyectada (un bit del segmento de código o del de datos); es decir, se inyectan fallos de tipo *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.

- En [Benso *et al.* 1998c, Benso *et al.* 1998d] se presenta una herramienta que utiliza las excepciones del procesador para realizar la inyección. En [Benso *et al.* 1998d], se da nombre a la herramienta: **EXFI** (*EXception-based Fault Injector*). La aplicación de EXFI se realiza sobre una placa comercial M68KDIP de Motorola, con un microprocesador M68040. En [Benso *et al.* 1998d] se presentan los resultados de tres campañas de inyección realizadas (con tres cargas distintas: una implementación del algoritmo *bubblesort*, un analizador sintáctico, y una multiplicación de matrices de 10×10) con el fin de comprobar la funcionalidad de la herramienta. Como conclusión de la técnica se deduce que es un método de 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** [Benso *et al.* 1999b], que permite inyectar fallos mediante SWIFI utilizando tres métodos de inyección diferentes:

- ⇨ Utilizando excepciones [Benso *et al.* 1998c, Benso *et al.* 1998d].
- ⇨ Híbrido [Benso *et al.* 1998b]. 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 [Benso *et al.* 1999a]. Los nuevos microprocesadores y microcontroladores producidos por Motorola disponen de un modo de funcionamiento especial, denominado modo de diagnóstico (*Background Debug Mode*, BDM).

- **UMLinux** (*User Mode Linux*) [Sieh y Buchacker 2002, Höxer *et al* 2002] 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 [Sieh 1993]. En [Höxer *et al* 2002] 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) [Yuste *et al.* 2003] es una herramienta realizada por el Grupo de Sistemas Tolerantes a Fallos de la Universidad Politécnica de Valencia. Ha sido desarrollada bajo los auspicios del proyecto de investigación europeo DBench [DBench 2003] (IST-2000-25425), para la inyección de fallos en sistemas empotrados de tiempo real. Esta herramienta se basa en Nexus [Nexus 1999], un 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. Se trata, pues, de una herramienta híbrida.
- Otros resultados:
  - ⇨ **CSFI** (*Communication Software Fault Injector*) [Carreira *et al.* 1995] es una herramienta producida en el proyecto europeo FTMPS Nº 6731 con el fin de inyectar fallos de comunicaciones en computadores paralelos. En [Carreira *et al.* 1995] 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** [Dawson *et al.* 1996a, Dawson *et al.* 1996b] se ha desarrollado en la Universidad de Michigan, para su aplicación a sistemas distribuidos en tiempo real. ORCHESTRA se implementa como un inyector de fallos a nivel de protocolo, instalándolo entre dos capas del protocolo de comunicaciones. La misión de esta nueva capa de inyección de fallos (*Protocol Fault Injector* –PFI– *layer*) es interceptar y manipular los mensajes que pasan a su través. En [Dawson *et al.* 1996a, Dawson *et al.* 1996b] se muestran los resultados de la aplicación de ORCHESTRA a dos sistemas (protocolos): una aplicación de video-conferencia en tiempo real sobre Real-

Time Mach y servicio de pertenencia a grupo sobre un sistema operativo Sun Solaris.

- ⇨ En [Fuchs 1996] 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 el 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 a nivel de aplicación y la existencia de violaciones de la asunción de silencio en caso de avería (en inglés *fail-silence*).
- ⇨ Otra aplicación interesante se presenta en [Stott *et al.* 1997, Stott *et al.* 1998]. En este caso 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 [Stott *et al.* 1997, Stott *et al.* 1998] se muestran los resultados de dicha aplicación. En [Stott *et al.* 1998], 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 3.2.5). La comparación da un 83% de concordancia en los comportamientos producidos en ambos casos. La elevada discrepancia se justifica por que el modelo realizado para la inyección mediante simulación no es completo.
- ⇨ Un ejemplo de aplicación a sistemas empotrados se puede ver en [Velazco y Rezgui 2000, Velazco *et al.* 2000]. En estos trabajos se presenta la herramienta **THESIC** (*Testbed for Harsh Environment Studies on Integrated Circuits*). THESIC 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 (*Memory Mapped Interface*, MMI). Por cada sistema que se analice con THESIC será necesario construir una placa secundaria. En [Velazco y Rezgui 2000, Velazco *et al.* 2000] se presentan los resultados de aplicar THESIC a dos sistemas: un microcontrolador 80C51 (ejecutando una multiplicación de 6×6) y un DSP (*Digital Signal Processor*) TI 320C50 (ejecutando una aplicación de módem).

3.5 Comparación de las técnicas de inyección de fallos

La caracterización de una técnica de inyección de fallos depende de un conjunto de criterios de orden científico (objetivos de la experiencia), técnico (fase del ciclo de vida, tecnología utilizada, etc.) y económico (coste).

En [Jenn 1994a] se utilizan seis criterios como base para comparar las técnicas de inyección (física, SWIFI y simulación):

- La genericidad, que caracteriza la independencia de la técnica respecto del sistema.
- La accesibilidad y la controlabilidad, que tienen en cuenta la facultad de acceso a los puntos de inyección, y la capacidad de control de los atributos de los fallos inyectados.

- La neutralidad, que mide la importancia de las modificaciones inducidas por la técnica de inyección sobre el sistema. Otros autores utilizan como medida la *intrusividad* [Hsueh *et al.* 1997], que es el concepto opuesto a la neutralidad.
- La automatización.
- La precisión, que permite evaluar la adecuación de la técnica a las hipótesis de fallos considerados en los objetivos de la campaña.
- El coste.

3.5.1 Genericidad

La genericidad de las técnicas de inyección mediante simulación depende de la del formalismo de modelado. Si la larga gama de formalismos y herramientas de inyección mediante simulación permite la aplicación de la inyección de fallos a todos los estados de desarrollo, la utilización de una misma técnica en el curso de numerosas fases de desarrollo no depende más que de la expresividad del lenguaje de modelado. La tendencia actual está en la búsqueda de formalismos que cubran un gran conjunto de modelos, lo que aumenta considerablemente la genericidad de las técnicas de inyección asociadas a estos formalismos.

La utilización de técnicas HWIFI está condicionada por la existencia de un modelo físico (prototipo o versión operacional del sistema), y su aplicabilidad está limitada a las fases finales del desarrollo. Sin embargo, presenta el interés de permitir la validación del sistema real completo, incluyendo el *hardware* y el *software*.

Por otra parte, a cada una de las técnicas de inyección implementada mediante *hardware* se le asocia un dominio de aplicación, limitado por restricciones de orden físico tales como:

- La encapsulación (de cerámica, plástico, etc.) y/o los terminales.
- La naturaleza del substrato del circuito integrado. Por ejemplo, la insensibilidad a los iones pesados de los circuitos CMOS sobre substrato aislante (SOI, del inglés *silicon-on-insulator*).
- La disipación térmica (dificultades para enfriar el circuito cuando está situado en el recinto en vacío requerido para la inyección de iones pesados o, a la inversa, dificultad para acceder al circuito para proceder a la inyección a nivel de los terminales cuando está situado en un recinto de refrigeración).
- El proceso de fabricación de un circuito (imposibilidad de utilizar la técnica de rayo láser si los transistores están situados bajo la capa metálica, por ejemplo).

Las técnicas de inyección implementadas mediante *software* (SWIFI) tienen una limitación similar, puesto que se aplican sólo a los sistemas que ejecutan programas. Estas técnicas no son aplicables más que al final del diseño, cuando coexisten el *hardware* y el *software*; se trata, en efecto, de poder ejecutar los programas en su entorno operacional. La existencia de dependencias entre los mecanismos de inyección y su entorno (utilización de instrucciones específicas de un procesador y de mecanismos propios de un sistema de explotación) implica un elevado grado de especificidad de la herramienta de inyección SWIFI respecto de un sistema concreto. Este fenómeno tiene como consecuencia observable que el conjunto de herramientas presentadas en la bibliografía están asociadas a un sistema particular. Sin embargo,

de forma general, la adaptación de una técnica o de una herramienta de inyección por emulación de un entorno a otro es a priori una tarea fácil.

3.5.2 Accesibilidad y controlabilidad

La inyección mediante simulación presenta en teoría una accesibilidad absoluta, puesto que el modelo es una estructura de datos. En realidad, la única limitación viene dada por la accesibilidad del entorno de simulación sobre el modelo. Los simuladores comerciales permiten normalmente acceder a cualquier señal o variable del modelo. Por otro lado, la reproducibilidad de una experiencia está generalmente asegurada: basta con guardar el conjunto de datos que caracterizan el estado del simulador en el momento de la inyección con el fin de poder reutilizarlo para iniciar una experiencia posterior.

La accesibilidad en las técnicas de inyección implementadas mediante *hardware* depende de la relación entre el lugar donde se inyecta y la técnica: la estructura interna de un circuito integrado encapsulado es inaccesible a los medios utilizados para efectuar inyecciones a nivel de *pin* (utilizando sondas o soportes especiales), pero es totalmente accesible a técnicas de inyección por radiación. De forma general, la reproducibilidad es extremadamente reducida, puesto que reposa en parte sobre la controlabilidad (es decir, la facultad de poder especificar los atributos de los fallos) y sobre la facultad de situar al sistema en un estado determinado. Es de destacar en este caso que la inyección de fallos a nivel de *pin* presenta una controlabilidad muy superior a la del resto de las técnicas HWIFI.

En la inyección SWIFI de fallos físicos, se trata de situar al sistema en un estado preciso, correspondiente a la ocurrencia de un fallo físico particular. En el caso de un procesador, el límite en la accesibilidad viene dado, por ejemplo, por la imposibilidad de definir el contenido de ciertos registros internos a través del juego de instrucciones, de situar dos señales tales como *READ* y *WRITE* en un estado incoherente (*READ* y *WRITE* simultáneamente activas) o de realizar operaciones sobre el *bus* no sincronizadas con el reloj. Desde el punto de vista de los mecanismos de emulación lógica, accesibilidad y controlabilidad tienen significados idénticos, pues para acceder a diferentes puntos de inyección sólo se utilizan los mecanismos estándar del *software*.

3.5.3 Neutralidad

Aunque la neutralidad de las técnicas de inyección difiere sensiblemente de una técnica a otra, es en general **escasa**. En efecto, estas técnicas requieren un acceso más o menos directo a los puntos de inyección.

En la inyección implementada mediante *hardware*, este hecho se hace especialmente evidente en la inyección a nivel de *pin* por inserción (el circuito debe poder ser extraído de la tarjeta sobre la que se encuentra, y esto puede ser un problema si el circuito está soldado) y en el bombardeo con iones pesados (el circuito debe estar situado en un recinto especial). La aplicación de una sonda (caso de la inyección a nivel de *pin* por forzado) o el alargamiento de las conexiones (caso de la inyección a nivel de *pin* por inserción) traen consigo modificaciones de las características eléctricas del sistema, susceptibles de engendrar mutaciones parásitas o fallos permanentes. En el caso de la inyección por bombardeo de iones pesados, en particular, se pueden generar fallos permanentes como consecuencia de fenómenos de *latch-up*.

En la inyección mediante simulación, en general, el modelo no está sujeto a la ocurrencia de mutaciones parásitas no descubiertas, ni a la introducción de fallos duros. En el caso parti-

cular de la emulación de fallos con FPGA, hay que añadir cambios en el modelo para poder realizar la inyección.

Las modificaciones necesarias para la implementación de la inyección SWIFI tienen lugar en los programas de la aplicación y/o en los de explotación. Estas modificaciones pueden ser el origen de mutaciones parásitas. La inserción de instrucciones específicas de la inyección modifica la duración de la ejecución del código. Sin embargo, como en el caso de la inyección por simulación, estas técnicas impiden la introducción de fallos permanentes en el sistema.

3.5.4 Automatización

Un modelo de simulación es naturalmente más fácil de tratar que un modelo físico, ya que la determinación de las características de los atributos de las experiencias de inyección, la colocación de los inyectores y el tratamiento de los resultados de la inyección, sólo necesitan manipulaciones de datos informáticos. Caso aparte es la emulación de fallos con FPGA, donde hay que adaptar el modelo para poder aplicarle la inyección y recompilarlo, que son tareas (hasta el momento) difíciles de automatizar.

La necesidad de desplazar la sonda de inyección (necesaria en la inyección a nivel de *pin* por inserción, por ejemplo) limita las posibilidades de automatización de ciertas técnicas HWIFI. Este no es el caso para todas estas técnicas; en particular, la inyección por bombardeo de iones pesados es automatizable en gran medida. De forma general, las demás operaciones (inyección, medidas, inicialización del sistema, etc.) pueden ser efectuadas de forma automática por el banco de test.

En las técnicas de inyección SWIFI, se manejan tipos de datos similares a los manejados por las técnicas de simulación, por lo que la capacidad de automatización de las herramientas SWIFI pueden considerearse similares.

3.5.5 Precisión

La precisión que se puede obtener por medio de una técnica de inyección mediante simulación está limitada, por una parte, por la capacidad de modelado de fallos del formalismo y, por otra, por la extensión de los dispositivos de inyección que ofrece. De manera general, se puede afirmar que la correspondencia entre los modelos de fallos físicos y los fallos inyectados se establece preferentemente en los niveles de modelado más bajos.

Desde el punto de vista de la precisión, la inyección implementada mediante *hardware* **favorece** naturalmente la equivalencia, directa o indirecta, entre los fallos inyectados y los reales encontrados durante la vida operacional del sistema.

Esta equivalencia es prácticamente absoluta en el caso de técnicas de inyección por radiación; en efecto, los fallos inyectados son de naturaleza similar a los engendrados por radiaciones iónicas encontradas en la atmósfera[27] (a nivel de componentes de procesadores situados en satélites, por ejemplo).

[27] Tradicionalmente, sólo se simulaban las condiciones de radiación en la atmósfera alta. En la actualidad, a causa de las más recientes previsiones acerca de cómo van a afectar las radiaciones a las generaciones de circuitos integrados actuales y futuras (véase el apartado 4.3.3), también empiezan a tenerse en cuenta las radiaciones en la atmósfera baja e incluso a nivel del mar.

En la inyección HWIFI a nivel de *pin*, la equivalencia es indirecta. Los fallos de rotura de pista o cortocircuito a la alimentación tienen tendencia a desaparecer gracias a la mejora del proceso de fabricación. Los fallos inyectados deben representar entonces los fallos internos a los circuitos. Si esta representatividad es corroborada en cierta medida por la experiencia, es sin embargo interesante constatar que los modelos de fallos de *stuck-at* inyectados en la periferia de los circuitos son los mismos en el caso de circuitos MSI (puertas lógicas) que en circuitos de más alta escala de integración; está claro que la precisión no puede ser la misma en ambos casos. De hecho, la hipótesis que sostiene la técnica de inyección a nivel de *pin* de que la mayoría de los fallos internos se producen a nivel de circuitos de interfaz, y se manifiestan como fallos *stuck-at* a nivel de *pin*, debe necesariamente ser revisada para tener en cuenta el aumento exponencial de la densidad de integración.

Desde el punto de vista de la precisión, las técnicas de inyección SWIFI presentan cierta similitud con las HWIFI. En efecto, la búsqueda de una equivalencia entre los efectos de los fallos inyectados y los de los fallos físicos se basa en los esfuerzos de modelado de estos efectos a nivel del *software*, pero también en las limitaciones propias de la aproximación, como la controlabilidad.

3.5.6 Coste

El coste no se tiene que ver como algo global, dado que se puede enfocar desde cuatro puntos de vista: la elaboración de la herramienta (incluyendo el diseño y la implementación), la preparación de los experimentos, la inyección de los fallos y el tratamiento de los datos obtenidos. Por otro lado, el concepto de coste se debe tener en cuenta tanto desde el punto de vista económico como desde el temporal.

Las técnicas de inyección basadas en modelos analíticos o en simulación sólo usan los programas de tratamiento de la información contenida en los modelos sobre los que se aplican. Por este motivo, el coste económico de la herramienta comprende esencialmente el coste del entorno de simulación. En el caso de la emulación con FPGA hay que tener en cuenta, además, el coste de la placa de desarrollo con la que se trabajará (en caso de que no se disponga de una adecuada). Esto debe sin embargo ser ponderado en el caso en el que el entorno sea independiente de la herramienta de inyección, y se utilice con otros fines, como el diseño del sistema. En cuanto al coste temporal asociado al desarrollo de la herramienta, éste es relativamente reducido, ya que en principio sólo hay que desarrollar una aplicación.

El coste (temporal) de la preparación de la campaña de inyección es esencialmente debido a la creación del modelo, que es una tarea independiente de la validación. En realidad, el factor determinante es, con creces, el *tiempo* necesario para la simulación del modelo; de hecho, sólo una simulación de alto nivel o una simulación de bajo nivel limitada a una parte reducida del modelo son en la práctica factibles.

El coste de la elaboración de las herramientas HWIFI es muy variable según la técnica. Las herramientas basadas en la inyección de iones pesados son posiblemente las más costosas. En la inyección por exposición es necesaria una fuente radiactiva (difícil de adquirir, tanto por su coste económico como por los especiales pasos que hay que seguir), y en la inyección mediante el bombardeo con un acelerador (siendo preciso alquilar la instalación, con el consiguiente coste económico, además de las restricciones temporales). En cuanto a las herramientas de inyección a nivel de *pin*, algunas de ellas están disponibles comercialmente, lo que en caso de resultar satisfactorias reducen a cero el coste de su realización. El resto de las subtécnicas se encuentran entre ambos extremos.

Los costes inherentes a la preparación, y a la inyección propiamente dicha, resultan de la preparación de las infraestructuras materiales (especialmente en el caso de la inyección con iones pesados utilizando un acelerador de partículas) y del propio sistema.

Por su naturaleza puramente lógica, el coste de una herramienta SWIFI es *escaso*: es el coste del soporte. Del mismo modo, los costes de preparación y de inyección son reducidos, sabiendo que los mecanismos de inyección son, en general, específicos a un procesador y un sistema de explotación dados. El coste de la inyección es idéntico al de la inyección HWIFI, ya que las dos técnicas son aplicadas al sistema físico.

Por último, ya que una campaña de inyección da lugar de forma casi sistemática a un conjunto de datos (almacenados durante las experiencias), el coste ligado a su tratamiento es generalmente el mismo para todas las técnicas de inyección, sean físicas o simuladas.

3.5.7 Conclusión

En la Tabla 3.1 se muestra de manera resumida la comparación de los tres grupos principales de técnicas de inyección en función de los criterios anteriores. Obsérvese que de la columna correspondiente a las técnicas basadas en simulación se ha extraído otra columna para la emulación con FPGA, puesto que presentan discrepancias importantes.

		Técnica			
		Simulación	**Emulación con FPGA**	**HWIFI**	**SWIFI**
Atributo	**Genericidad**	Alta	Alta	Media	Baja
	Accesibilidad y Controlabilidad	Alta	Alta	Media	Media
	Neutralidad	Alta	Baja	Baja	Media
	Automatización	Alta	Media	Media	Alta
	Precisión	Baja	Baja	Alta	Media
	Coste	Medio	Medio	Alto	Bajo

Tabla 3.1: Comparación de las técnicas de inyección.

3.6 La inyección de fallos en el futuro

A causa de una serie de factores, las herramientas de inyección de fallos desarrolladas en función de las técnicas tradicionales descritas anteriormente están siendo incapaces de utilizarse en los nuevos diseños.

Por un lado, las nuevas generaciones de circuitos integrados presentan una gran cantidad de *pines* (ya que los SoC (del inglés *Systems on a chip*) integran cada vez un mayor número de dispositivos), lo que afecta directamente a las técnicas HWIFI. El elevado número de *pines* hace que sea necesaria una gran cantidad de sondas en las herramientas de inyección a nivel de *pin*, y complica enormemente la utilización de SCIFI. Los procesadores más modernos tienen una cada vez mayor cantidad de circuitería inaccesible a través del *software*, lo que perjudica claramente a las técnicas SWIFI.

Por otro lado, las nuevas tendencias en el diseño de sistemas (sistemas distribuidos, redes de comunicaciones, sistemas en tiempo real, sistemas empotrados, etc.) no hacen sino acentuar las carencias de las técnicas y, por consiguiente, de las herramientas.

Esto, y el hecho de que ninguna de las técnicas de inyección es perfecta, ha hecho que en lugar de hacer competir a las técnicas, se las haga colaborar, realizando herramientas de inyección que permitan realizar inyección de fallos sobre diferentes tipos de sistemas, y siguiendo técnicas distintas, con una única interfaz de usuario. Además se pretende que las herramientas sean fácilmente portables.

A continuación se describen cuatro ejemplos de este cambio de actitud en los centros de investigación respecto a las técnicas de inyección de fallos:

- **LIVE** (*Low-Intrusion Validation Environment*) [Amendola *et al.* 1997]. Esta es una herramienta desarrollada en la empresa (Ansaldo-Cris) para inyectar fallos en sistemas de control ferroviario. Es un sistema que permite inyectar fallos mediante inyección HWIFI a nivel de *pin* por forzado y mediante SWIFI.
- **NFTAPE** [Stott *et al.* 2000]. Surge en la Universidad de Illinois como una ampliación de FTAPE [Tsai e Iyer 1995a, Tsai e Iyer 1995b]. 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 [Stott *et al.* 2000] 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** [Santos y Costa 2001]. Esta herramienta está siendo desarrollada a partir de Xception [Carreira *et al.* 1998]. 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 [Santos y Costa 2001] 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*) [Aidemark *et al.* 2001]. Esta herramienta ha sido desarrollada en Java, y utiliza bases de datos compatibles con SQL, lo que la hace altamente portable entre plataformas distintas. La implementación presentada en [Aidemark *et al.* 2001] permite inyectar fallos mediante Scan-Chain y SWIFI aplicada en tiempo de compilación (o antes de la ejecución, *pre-runtime SWIFI*). Al igual que las dos 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.

3.7 Resumen y conclusiones

En este capítulo 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*. 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 del test de *software*.

Se ha hecho una descripción de las características de cada una de ellas y se han mostrado algunas de sus subtécnicas más destacadas, así como algunos de los trabajos publicados más relevantes en cada caso, haciendo mención especial a las herramientas realizadas.

Al final de la exposición se han comparado los tres grupos de técnicas en función de seis criterios: genericidad, accesibillidad, neutralidad, automatización, precisión y coste (económico y temporal). La conclusión principal que se ha extraido es que, por dos motivos, ninguna de las técnicas es perfecta, y no deben considerarse como excluyentes o independientes, sino como complementarias.

El primer motivo de imperfección de las técnicas de inyección es que ninguna se puede aplicar en todas las fases del ciclo de vida de un sistema (evidentemente en las que se puede utilizar la inyección de fallos como método de validación). Antes al contrario, cada técnica sólo es útil en una fase. Así, las técnicas de inyección mediante simulación sólo se suelen aplicar en la fase de diseño, y las técnicas HWIFI y SWIFI en la fase de prototipado.

La segunda razón es que todas las técnicas presentan deficiencias en algunos de los seis criterios de comparación expuestos. En este sentido:

- Las técnicas de inyección mediante simulación ofrecen una precisión muy baja, debida a su aplicación a modelos. Como contrapartida, disponen de excelentes capacidad de automatización y genericidad, así como accesibilidad y controlabilidad totales.
- Las técnicas implementadas mediante *hardware* adolecen de un elevado coste, tanto económico en su elaboración como temporal en la preparación de los experimentos. Además, son técnicas bastante intrusivas (con baja neutralidad). En cambio, la precisión de sus resultados es la mejor de todas.
- Las técnicas SWIFI tienen muy bajos costes de elaboración (tanto desde el punto de vista económico como del temporal) y aplicación, pero disponen de poca accesibilidad y controlabilidad, y sobre todo de muy baja genericidad.

Reafirmando las conclusiones anteriormente expuestas, en los últimos años se está dando una nueva tendencia en el desarrollo de herramientas de inyección de fallos, que recoge la problemática mencionada y la reconduce hacia la integración, en un mismo prototipo, de diferentes subtécnicas. 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.

4 Modelos de fallos

4.1 Introducción

En un sistema se pueden distinguir diferentes niveles de abstracción, en función de quién y cómo se interrelacione con él. En [DBench 2002] se consideran ocho niveles: físico-electrónico, lógico, transferencia de registro (RT), algorítmico, *kernel*, *middleware*, aplicación y operación.

Ésta no es la única clasificación existente, y de hecho existe un gran número de autores que proponen otras clasificaciones con diferente número y nombre de niveles[28]. No obstante, todos ellos están relacionados y a menudo solapados entre sí.

En cada nivel se pueden producir fallos, que a su vez pueden ocasionar errores que el usuario del sistema a ese nivel (o a un nivel superior) observa como averías.

Es muy importante establecer para cada nivel un conjunto de modelos de fallos que reflejen lo más fielmente posible los efectos de los fallos reales, tanto en el efecto producido en el sistema como en su temporización.

La innumerable cantidad de fallos físicos, junto con su elevada complejidad, justifican la necesidad de modelar dichos fallos (o sus efectos) en los diferentes niveles de un sistema informático donde afecta.

La ventaja de la modelización es que un solo modelo puede englobar diversos fallos físicos, simplificando de esta forma su tratamiento. En el contexto particular en que se enclava el presente trabajo de tesis, la utilización de modelos de fallos reduce el coste de la simulación de la inyección de los fallos. El inconveniente es la gran dificultad existente para construir modelos que reflejen fielmente el comportamiento físico de los circuitos integrados.

Dichos modelos tienen la ventaja de combinar varios fallos físicos en un único modelo de fallo, reduciendo el número de fallos distintos que se pueden introducir en el sistema, simplificando la inyección de fallos tanto temporal como espacialmente.

Los fallos *hardware* abarcan los niveles físico-electrónico, lógico y RT, como muestra la Figura 4.1. Los fallos *hardware*, y en particular la representatividad de los modelos de fallos en los niveles lógico y RT, son muy importantes en el contexto de este trabajo de tesis, porque es precisamente en estos niveles donde las diferentes técnicas basadas en la inyección de fallos sobre modelos en VHDL permiten inyectar fallos.

El objetivo de este capítulo será la deducción de un conjunto de modelos de fallo (en los niveles lógico y RT) que se pueden aplicar a las diferentes técnicas de inyección de fallos basadas en simulación en VHDL. Al mismo tiempo, se pretende que los modelos sean representativos de los fallos físicos reales producidos en los circuitos integrados VLSI actuales y futuros.

[28] Por ejemplo, la clasificación de [Walker y Thomas 1985] incluye cinco niveles: de circuito, lógico, de bloque funcional, algorítmico y de arquitectura. En [Siewiorek y Swarz 1992] se consideran tres niveles: de circuito, lógico y de sistema. En [Jenn 1994] se especifican más niveles: tecnológico, de circuito, lógico, RT y no interpretado. Por último, en [Pradhan 1996] se consideran tres niveles: eléctrico, lógico y funcional.

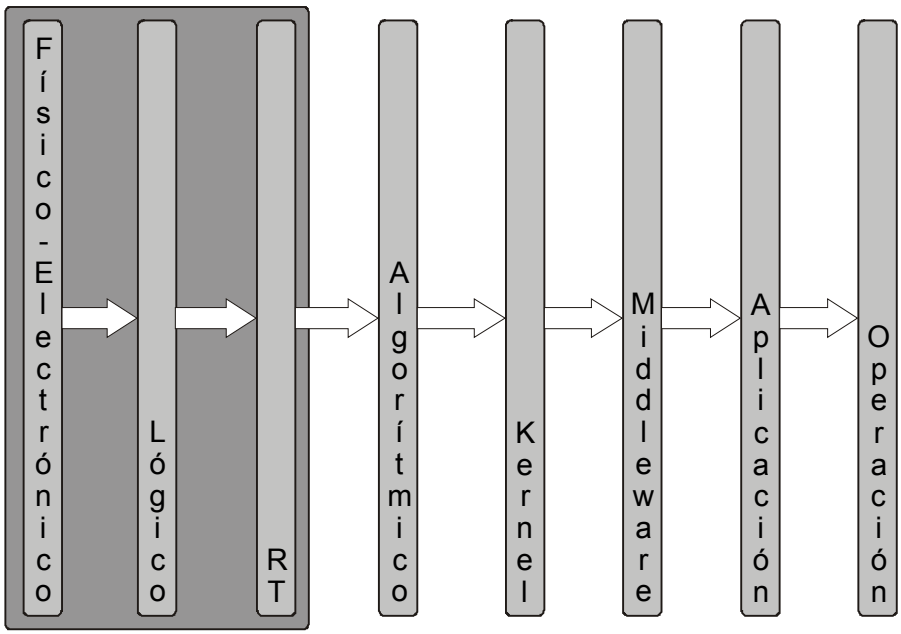

Figura 4.1: Niveles en los que se modelan los fallos *hardware*.

Por ello, es importante comprender la fenomenología física relacionada con los fallos para generar modelos eficientes en niveles de abstracción más elevados. Para estudiar la representatividad de los fallos en los niveles utilizados, es conveniente seguir una metodología "de abajo hacia arriba" (en inglés, *bottom-up*), basada en un conocimiento detallado de la causa primaria de los fallos. Los modelos de los fallos se deducen a partir de las causas físicas y los mecanismos implicados en la ocurrencia de fallos. Esta metodología está relacionada con los denominados "métodos de la física de las averías[29]" (en inglés, *Physics of Failure Methods*). Aunque bastante recientes, y complejos en su desarrollo, los métodos de análisis físico del fallo constituyen el futuro de la Fiabilidad [Amerasekera y Najm 1997].

Dependiendo de su duración, los fallos se clasifican en:

- Transitorios, con una duración corta en el tiempo.
- Permanentes, que perduran de manera indefinida.
- Intermitentes, similares a los transitorios en cuanto a que tienen una duración temporal, pero que se repiten en el tiempo sin un comportamiento periódico.

En este capítulo se estudiarán modelos para fallos permanentes, intermitentes y transitorios. Los modelos de los fallos permanentes están más establecidos en la bibliografía, mientras que los correpondiente a los fallos transitorios son un tema más abierto. La razón de esta diferencia es que los fallos permanentes se deben en su mayor parte a defectos de fabricación y al desgaste por su funcionamiento, aspectos que llevan asociada una localización espacial. Por el contrario, los fallos transitorios se presentan ilocalización espacial y son ocasionados por numerosas causas, especialmente externas. A esta dificultad se añade su elevada incidencia en los sistemas digitales. En cuanto a los fallos intermitentes, se ha considerado un con-

[29] Tradicionalmente, estos métodos se aplican exclusivamente al estudio de los fallos permanentes y sus consecuencias: las averías. En la actualidad, los fallos transitorios están siendo cada vez más considerados, y para su estudio pueden aplicarse las mismas técnicas que para los permanentes.

junto de modelos relacionados con los de los fallos permanentes, debido a que las causas y mecanismos son similares. De hecho, un elevado porcentaje de los fallos intermitentes aparecen como preludio de fallos permanentes.

Se mostrará la existencia de modelos no utilizados habitualmente, pero que cada vez proliferan más en los circuitos integrados VLSI. Se hará especial hincapié en las tecnologías MOS y CMOS, a la sazón las de mayor proyección.

En los siguientes apartados se estudia la representatividad de los modelos de fallos existentes en los dos niveles estudiados: lógico y RT. El apartado 4.2 muestra un resumen de diferentes modelos de fallos, en relación con los mecanismos físicos que los originan, y distinguiendo entre fallos permanentes, intermitentes o transitorios. En el apartado 4.3 se realiza un estudio de las nuevas tendencias en tecnologías submicrónicas de los circuitos integrados VLSI. En este apartado se profundizará en los modelos expuestos en el apartado 4.2, y se introducirán otros nuevos. Por último, en el apartado 4.4 se hace una recapitulación del trabajo realizado y de las conclusiones más importantes que se han extraído, dejando abiertas algunas líneas de actuación para trabajos futuros.

4.2 Mecanismos de fallo y modelos en los niveles lógico y RT

4.2.1 Introducción

Tradicionalmente, los únicos modelos de fallos considerados han sido *stuck-at*[30] (‘0’ y ‘1’) para los fallos permanentes, y *bit-flip*[31] para los fallos transitorios. Sin embargo, las novedades introducidas en el proceso de fabricación de los modernos circuitos integrados (referidas a metodologías de diseño, tecnologías de fabricación, y materiales utilizados) están provocando que el uso exclusivo de dichos modelos en la inyección de fallos no sea representativo de la casuística real.

Por este motivo, se hace necesaria la implantación de nuevos modelos que cubran los mecanismos predominantes en los circuitos integrados actuales, y que sirvan también para futuras familias tecnológicas.

En los siguientes subapartados se resumen 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 [Amerasekera y Najm 1997, Gil 1999, DBench 2002].

4.2.2 Fallos permanentes

Los fallos permanentes se deben a defectos irreversibles en los circuitos. Estos defectos pueden producirse durante el proceso de fabricación o por desgaste (en inglés *wear-out*) debido a su uso (ver Figura 4.2). En este último caso, algunos mecanismos por desgaste pueden en ocasiones manifestarse como fallos intermitentes hasta que provocan el fallo permanente. Por otro lado, los fallos debidos al proceso de fabricación pueden originar a su vez, fallos durante el funcionamiento del circuito. En la figura, esta relación se muestra con una línea discontinua.

[30] Pegado-a.
[31] Conmutación del valor del bit.

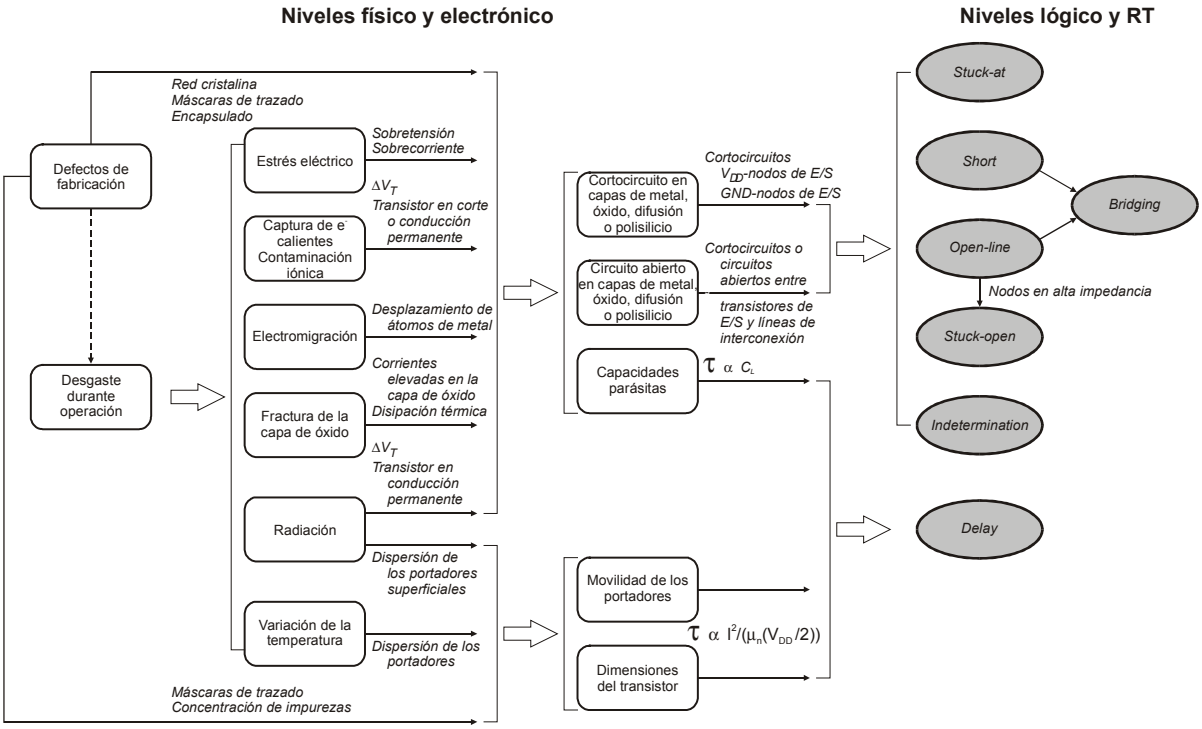

Figura 4.2: Mecanismos de fallos permanentes y modelos de fallo equivalentes [DBench 2002].

En la Figura 4.2 se pueden observar los modelos de fallos deducidos para los fallos permanentes (en los óvalos), así como los mecanismos físicos que los originan (en los rectángulos). En los arcos de la figura se indican algunas causas conocidas y establecidas que provocan los fallos [Siewiorek y Swarz 1992, Amerasekera y Najm 1997, Pradhan 1996, DBench 2002]. Además de los defectos de fabricación (en la red cristalina, en la definición de las máscaras de trazado[32], en el proceso de encapsulamiento, etc.), también se muestran algunas causas íntimamente ligadas al desgaste (estrés eléctrico, captura de electrones "calientes", rotura de la capa de óxido, electromigración, etc.). 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.). En función de su efectividad, estos fallos se pueden manifestar en el transistor en forma de conducción/corte permanente[33] o como los clásicos cortocircuito/circuito abierto. Estos fallos en las capas del transistor pueden originar fallos de cortocircuito entre las líneas de alimentación (V_{DD} y GND) y los nodos de entrada/salida del circuito lógico. También pueden ocasionar fallos de cortocircuito/circuito abierto tanto en los transistores de entrada/salida como en las líneas de conexión entre circuitos lógicos. 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 modelo más comúnmente utilizado.
- ***Open-line*** (circuito abierto, o alta impedancia) en las líneas de conexión de circuitos lógicos.

[32] En inglés, *layout*.
[33] En inglés, *stuck-on/stuck-off*.

- ***Short*** (cortocircuito)[34] entre las líneas de conexión de circuitos lógicos.
- ***Bridging*** (puente)[34], 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 [Pradhan 1986]), 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 [Gil 1999]. 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).

4.2.3 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.

4.2.4 Fallos transitorios

Estos fallos, también denominados *soft errors* y *single event upsets* (SEU), pueden aparecer durante el funcionamiento de un circuito por diferentes causas, tanto internas como externas. A diferencia de los permanentes, los fallos transitorios no introducen ningún defecto físico en el circuito.

El tratamiento de estos fallos es complejo, porque no se pueden ubicar espacialmente y por su corta duración. Además, no tienen un modelo bien definido debido a la variedad de fenómenos locales y externos que los pueden originar. Otro aspecto que realza su importancia es que la mayoría de los fallos en circuitos integrados digitales son transitorios. Por estas razones, el diseño de sistemas tolerantes a los fallos transitorios es una tarea muy complicada.

En la Figura 4.3 se pueden observar los modelos de fallos deducidos para los fallos transitorios. Como se puede ver, los modelos ***bit-flip*** (aplicado a elementos de almacenamiento: memorias y registros), ***pulse***[35] (aplicado a circuitos combinacionales) e ***indetermination*** per-

[34] Aunque tradicionalmente el nombre de este modelo hace referencia a un cortocircuito [Pradhan 1986, Shaw *et al.* 2001], algunos autores consideran como puente ciertas combinaciones entre cortocicuitos y circuitos abiertos que dan lugar a fallos más complejos [Jenn 1994, Gil 1999], y al modelo considerado tradicionalmente como puente se le denomina cortocircuito (*short*). Esta es la idea que se ha seguido en este trabajo de tesis, a excepción del apartado 3.3.1.1, donde se hace referencia al modelo *bridging* con su significado tradicional.

[35] Pulso.

miten 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[36]), 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) [Amerasekera y Najm 1997], 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*).

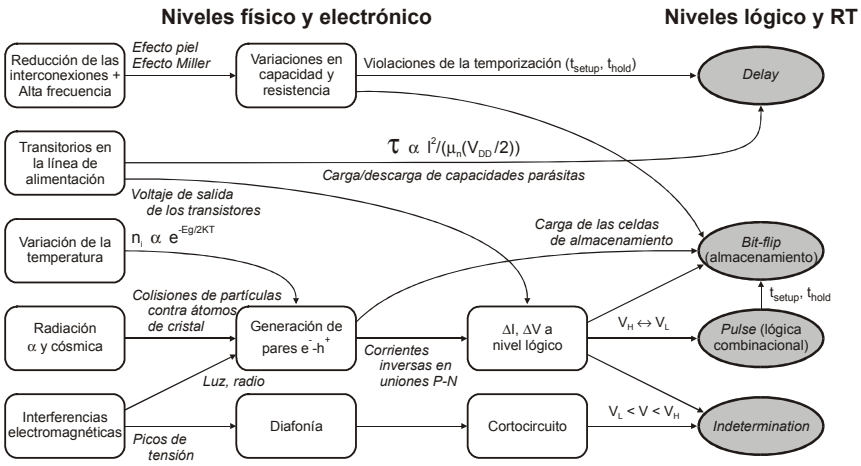

Figura 4.3: Mecanismos de fallos transitorios y modelos de fallo equivalentes [DBench 2002].

Aunque aparentemente los modelos *bit-flip* y *pulse* se corresponden con un mismo modelo, no es así. Ambos modelos coinciden en cuanto al modo en que se produce el fallo (invirtiendo el valor lógico original), pero se diferencian en lo que le ocurre al sistema cuando el fallo desaparece. En los elementos de almacenamiento (registros y memorias), el valor erróneo permanece hasta que se rescribe un nuevo valor. En cambio, en los circuitos combinacionales, al desaparecer el efecto del fallo, el valor erróneo es sustituido "inmediatamente" por el correcto. Es decir, la señal presenta la forma de un pulso, que es lo que da nombre al modelo.

A su vez, un pulso en la lógica combinacional puede afectar a registros o celdas de memoria en etapas posteriores, al ser almacenado si su duración se solapa con los márgenes temporales ($t_{set\text{-}up}$, t_{hold}, etc.). Eso daría lugar a un *bit-flip* o a otras manifestaciones menos frecuentes que se comentarán en el apartado 4.3.3.

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

El modelo ***delay*** permite representar fallos físicos debidos a transitorios en la alimentación (V_{DD}), que pueden alterar el retardo de conmutación ($\tau \propto l^2/(\mu_n(V_{DD}/2))$) de los transistores MOS o los tiempos de carga/descarga de las capacidades parásitas de las conexiones de entrada/salida.

En la Figura 4.3 también se incluyen algunos mecanismos debidos a las tendencias actuales y futuras de los circuitos submicrónicos. Los factores principales de fallo son la extremada reducción de la sección y separación de las interconexiones y las elevadas frecuencias de funcionamiento, que hacen que los efectos "piel" (en inglés *skin effect*) y Miller provoquen alteraciones de la constante de tiempo RC. Éstas, a su vez, pueden originar violaciones de los márgenes de tiempo (principalmente $t_{set\text{-}up}$ y t_{hold}) que derivan en la corrupción de los datos almacenados en los registros. En el apartado 4.3.3.2 se hace referencia de nuevo a estos mecanismos.

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

4.3 Influencia de las nuevas tecnologías submicrónicas en los mecanismos y modelos de fallos en los niveles lógico y RT

La Confiabilidad de los circuitos VLSI es un aspecto que cada vez tiene más importancia, ya que la utilización de este tipo de dispositivos en sistemas fiables e incluso de seguridad crítica (en inglés *critical-safety systems*) se ha incrementado considerablemente. Los avances introducidos en las últimas décadas tanto en las metodologías de diseño como en las tecnologías de fabricación de semiconductores han incrementado de manera vertiginosa la potencia de los sistemas basados en computadores. Sin embargo, la reducción de las geometrías, la disminución de las tensiones de alimentación y las elevadas frecuencias de funcionamiento tienen un impacto negativo en la Confiabilidad, ya que han incrementado las tasas de los fallos permanentes, intermitentes y transitorios.

A continuación se muestra el estudio del impacto de las nuevas tecnologías submicrónicas en los mecanismos físicos de fallo. Asimismo, se relacionan dichos mecanismos con modelos de fallo en los niveles lógico y RT. Para ello se ha tomado como base la relación entre mecanismos y modelos mostrada en el apartado 4.2. El estudio sigue la estructura natural presentada en el apartado, y se muestran por separado los análisis relativos a los fallos permanentes, intermitentes y transitorios, centrando el estudio en los mecanismos de fallo que tienen especial incidencia en los modernos circuitos submicrónicos.

4.3.1 Fallos permanentes

Los grupos de mecanismos de fallo más significativos en las nuevas tecnologías son: daños en la capa de óxido, metalización y encapsulado y ensamblado. Para cada grupo se especificarán los mecanismos particulares de fallo, describiendo para cada uno el problema que produce y sus efectos; por último, se le asociará(n) el(los) correspondiente(s) modelo(s) de fallo en los niveles lógico y RT.

4.3.1.1 Daños en la capa de óxido

De entre los diferentes mecanismos de fallo relacionados con el deterioro de la capa de óxido, cabe destacar la rotura de la capa de óxido (en inglés *oxide breakdown*), la inyección de portadores calientes (en inglés *hot carrier injection*) y los daños producidos por procesos con plasma.

4.3.1.1.1 Rotura de la capa de óxido (*thin oxide breakdown*)

Descripción

El deterioro de la capa de óxido de la puerta es uno de los mayores problemas en las tecnologías MOS. Como consecuencia de ello, el mantenimiento de la integridad de la capa de óxido de la puerta (en inglés *Gate Oxide Integrity*, GOI) es extremadamente importante en el proceso de fabricación, ya que esta cuestión es uno de los factores clave que determinan el espesor de la capa de óxido de la puerta, porque las capas más delgadas son normalmente más sensibles al desgaste y al daño para una tensión de alimentación dada. Por tanto, los requisitos de la integridad de la capa de óxido tienen un papel preponderante a la hora de definir la máxima tensión de alimentación a la que pueden funcionar los circuitos diseñados en una determinada tecnología.

El espesor de la capa de óxido se reduce a medida que lo hacen las geometrías de los dispositivos:

- En 1978, $t_{ox} \approx 750$ Å
- En 1988, $t_{ox} \approx 250$ Å
- En 1998, $t_{ox} \approx 80$ Å
- En 2002 $t_{ox} \approx 25$–30 Å, que no está muy lejano del límite físico del óxido de silicio (SiO_2), impuesto por el efecto túnel.

Los mecanismos relacionados con la rotura de la capa de óxido, y los problemas implicados son numerosos [Hu y Lu 1999, Stathis 2001]. Básicamente, la rotura es un proceso que se produce en dos fases:

- Desgaste: Generación de defectos en el interior del óxido y en la interfaz debidas a corrientes a través de las capas de óxido (denominados *traps*). Las corrientes en el aislante son debidas al efecto túnel (Fowler-Nordheim o directo).
- Rotura: Los defectos aumentan, originando densidades de corriente localmente elevadas, seguidas por una elevada disipación térmica que puede destruir la capa de óxido.

Algunas aproximaciones para el modelado de los mecanismos de fallo se basan en la Teoría de la Percolación aplicada al desgaste y rotura. También se han introducido nuevos modelos para óxidos ultrafinos[37] ($t_{ox} < 40$ Å), como por ejemplo "V-model", donde un electrón debe atravesar todo el espesor de la capa de óxido antes de impactar contra la interfaz opuesta [Hawkins 2000].

[37] *Ultrathins*.

El mecanismo de desgaste de la capa de óxido, o rotura del dieléctrico dependiente del tiempo (en inglés *Time-Dependent Dielectric Breakdown*, TDDB), sucede en las partes débiles de la película de óxido a causa de defectos en la misma. La búsqueda de la integridad de la capa de óxido de la puerta se dirige hacia la reducción de esos defectos. Éstos son habitualmente debidos a la presencia de impurezas en el proceso de crecimiento óxido (*thermally grown oxide*) o de enlaces rotos en el óxido de silicio. Se han observado mejoras en la integridad de la capa de óxido de la puerta mediante la optimización del preproceso de la oxidación (*gate oxidation pre-clean*) y el proceso de pasivación sobre el conductor de la puerta [Strong *et al.* 1993].

Efectos

El deterioro de la capa de óxido puede resultar en una excesiva corriente de fuga en los pines de entrada y salida, en el incremento de la disipación de potencia en el circuito integrado y en la disminución de la velocidad del circuito. La corriente puede producir daños tanto en la interfaz como en el volumen (en inglés, *bulk*) del óxido, generando a su vez más defectos. Con el tiempo, el número de defectos puede llegar a ser lo suficientemente alto como para que la elevada corriente en el óxido cause una excesiva disipación térmica y un daño irreversible.

Los efectos de la rotura del óxido afectan de manera particular a las celdas de memoria, en las que las fugas en la puerta podrían ocasionar la pérdida del dato almacenado. En las memorias EEPROM, cuyo funcionamiento se basa en la conducción a través de la capa de óxido de la puerta, las capas de óxido están sometidas a fuertes campos eléctricos, por lo que este tipo de circuitos son especialmente sensibles a las debilidades de las capas de óxido en la puerta.

Modelos

Además de los modelos ***short*** y ***open-line*** (véase la Figura 4.2), asociados con los daños "severos" en los transistores producidos en la fase de rotura, es preciso introducir otros modelos relacionados con el desgaste.

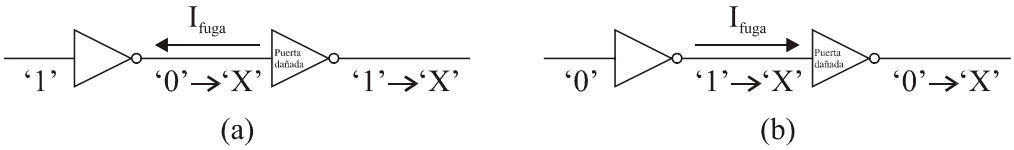

Figura 4.4: Generación de un valor de tensión indeterminado de salida por el aumento de la corriente de fuga en la puerta del transistor [DBench 2002]. (a) Perturbación de un valor lógico '0'. (b) Perturbación de un valor lógico '1'.

El aumento de la corriente de fuga (I_{fuga}) en la puerta del transistor puede incrementar la corriente de salida de las puertas, como se muestra en la Figura 4.4, afectando a los voltajes de salida, que pueden llevar a valores lógicos de tensión indeterminados. Por tanto, este tipo de fallo se puede representar con el modelo ***indetermination*** en la salida de la puerta.

Por otro lado, el aumento de la corriente de fuga puede afectar a la velocidad de conmutación de las puertas. La Figura 4.5 muestra los modelos de los tiempos de subida (τ_r) y bajada (τ_f) para un inversor CMOS. La puerta lógica carga o descarga, respectivamente, una carga capacitiva C_L. En el modelo para el tiempo de subida (Figura 4.5–a), se puede asumir que el transistor PMOS está en saturación durante todo el tiempo de carga [Pucknell y Eshraghian 1994], por lo que se modela como una fuente de corriente (I_{sat}). Por su parte, el transistor

NMOS está en corte. Se puede seguir un razonamiento análogo en el modelo para el tiempo de bajada (Figura 4.5–b): el transistor NMOS está en saturación y el PMOS está en corte.

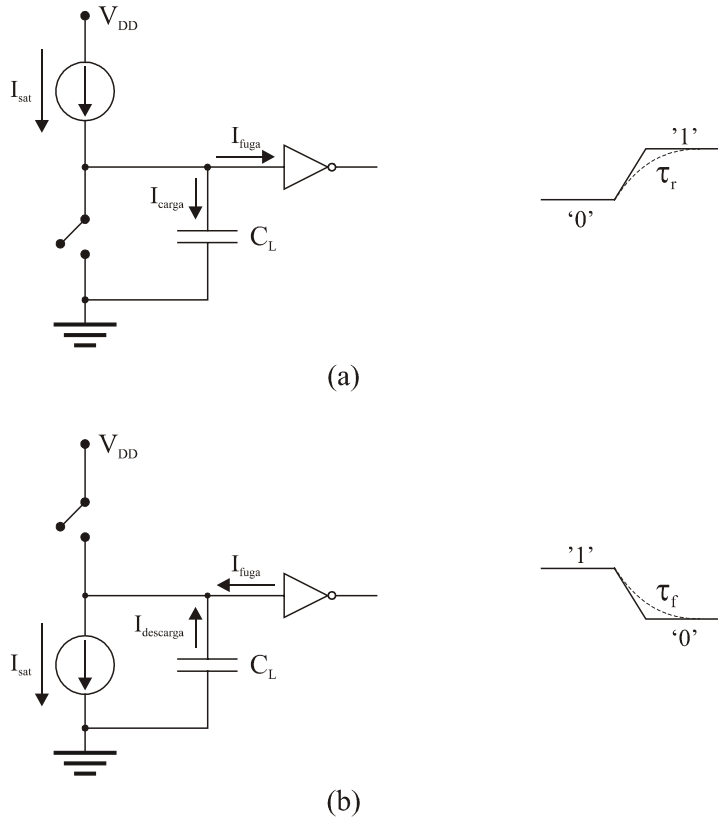

Figura 4.5: Efecto de las corrientes de fuga en las características temporales de las puertas [DBench 2002]. (a) Tiempo de subida. (b) Tiempo de bajada.

En ambos casos, el incremento de la corriente de fuga (I_{fuga}) debida a los defectos en la capa de óxido de la puerta puede modificar las corrientes netas de carga y descarga de C_L, disminuyendo sus valores, como se indica en las siguientes ecuaciones:

$$I_{carga} = I_{sat} - I_{fuga}$$

$$I_{descarga} = I_{sat} - I_{fuga}$$

De esta manera, los tiempos de carga y descarga aumentan, incrementándose los tiempos de subida y bajada. Este efecto se puede representar mediante el modelo ***delay***.

Otro efecto temporal relacionado con el aumento de las corrientes de fuga es la pérdida de datos, que se da en memorias (especialmente EEPROM) [Lee *et al.* 2001]. El valor del dato almacenado puede modificarse con el tiempo y perderse. El nuevo valor puede ser, bien el contrario al correcto (modelo ***bit-flip***), o tomar un valor indeterminado (modelo ***indetermination***).

4.3.1.1.2 Inyección de portadores "calientes" (*hot carrier injection*)

Descripción

Este mecanismo de fallo es extremadamente importante en los circuitos CMOS submicrónicos, en los que los problemas relacionados con los portadores "calientes" influyen de forma determinante en el diseño de los transistores [Rodder *et al.* 1995]. 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 (en inglés, *trapping*) de electrones y huecos en dicha capa. La captura de cargas es producida por el campo eléctrico lateral en el canal de los transistores MOS. Los portadores del canal (electrones en NMOS y huecos en PMOS) pueden alcanzar un nivel de energía que supere sus respectivas barreras de potencial en la interfaz Si-SiO_2. Los electrones tienen mayor movilidad, y energía cinética en el canal, por lo que tienen más probabilidad que los huecos de generar una carga capaz de deteriorar la capa de óxido. Es decir, el canal N es más vulnerable.

Efectos

El efecto de este mecanismo de fallo es una degradación gradual de algunos parámetros del transistor: la tensión umbral (V_T), la transconductancia (g_m) y la corriente del drenador (I_{DS}). El tipo concreto de degradación depende del tipo de transistor. En transistores NMOS, V_T aumenta y descienden g_m e I_{DS}. En transistores PMOS se produce una reducción de la longitud efectiva del canal del dispositivo, además de variaciones (positivas o negativas) de los tres parámetros indicados [Hawkins 2000].

A continuación se ve con más detalle el caso de los transistores NMOS. Como ya se ha comentado, la captura de electrones en la capa *thinox* disminuye la conductividad del canal, e incrementa V_T. Dada la relación de V_T con I_{DS} y g_m, expresada en las siguientes ecuaciones:

$$I_{DS} = K\left(V_{GS} - V_T\right)^2 \quad \text{(zona de saturación)} \tag{4.1}$$

$$I_{DS} = K\left[2\left(V_{GS} - V_T\right)V_{DS} - \left(V_{DS}\right)^2\right] \quad \text{(zona lineal)} \tag{4.2}$$

$$g_m = \left.\frac{\delta I_{DS}}{\delta V_{GS}}\right|_{VDS=constant} = 2K\left(V_{GS} - V_T\right) \tag{4.3}$$

éstas se ven, por consiguiente, alteradas. Por ejemplo, si V_T aumenta, (V_{GS}–V_T) disminuye, y en consecuencia, I_{DS} y g_m decrecen.

El efecto de la captura de huecos es el opuesto, es decir, la disminución de V_T. La causa es que las cargas positivas aumentan la conductividad del canal. Sin embargo, para que los huecos puedan moverse, tienen que superar una barrera de energía muy superior a la de los electrones, por lo que su movilidad es inferior.

Los modelos a nivel electrónico (transistor) son fallos *stuck-on* y *stuck-off* (es decir, el transistor se encuentra en conducción o corte permanente), que representan respectivamente pseudo-cortocircuitos y pseudo-circuitos abiertos entre el drenador y la fuente, con diferentes grados de efectividad en función de la conductividad del canal [Favalli *et al.* 1991].

Modelos

Los pseudo-circuitos abiertos y pseudo-cortocircuitos pueden causar la indeterminación de las tensiones de salida (modelo ***indetermination***), debido a que la resistencia de los transistores de salida tiene un valor intermedio: $R_{lineal} < R_{PMOS}, R_{NMOS} < R_{corte}$.

También las características temporales pueden verse afectadas por la alteración de la tensión umbral (V_T). De la Figura 4.5, y suponiendo que $I_{fuga} \approx 0$ (es decir, la puerta de carga no presenta fallos relacionados con la rotura de la capa *thinox*) y que $I_{IL} \approx 0$ (esta asunción es realista para las puertas CMOS), se pueden estimar los valores de los tiempos de subida (τ_r) y bajada (τ_f):

- **Estimación del tiempo de subida**

  La corriente de saturación para los transistores PMOS está determinada por:

$$I_{DSp} = K_p (V_{GS} + |V_{Tp}|)^2 \tag{4.4}$$

  Esta corriente carga C_L, y como su magnitud es aproximadamente constante, se tiene que:

$$V_{out} = \frac{I_{DSp} t}{C_L} \tag{4.5}$$

  Despejando t de la ecuación (4.5), y sustituyendo I_{DSp} por la expresión en la ecuación (4.4):

$$t = \frac{C_L V_{out}}{K_p (V_{GS} + |V_{Tp}|)^2} \tag{4.6}$$

  Si se asume que $t = \tau_r$ cuando $V_{out} = +V_{DD}$, entonces:

$$\tau_r = \frac{C_L V_{DD}}{K_p (V_{GS} + |V_{Tp}|)^2} \tag{4.7}$$

  Como $V_{GS} \approx -V_{DD}$, la expresión resultante es:

$$\tau_r \approx \frac{C_L V_{DD}}{K_p (V_{DD} - |V_{Tp}|)^2} \tag{4.8}$$

  El resultado aquí obtenido es razonablemente comparable a un análisis más detallado considerando por separado las zonas de saturación y lineal del transistor [Pucknell y Eshraghian 1994].

- **Estimación del tiempo de bajada**

  Haciendo asunciones y razonamientos similares a los del caso anterior, pero aplicándolos a un transistor NMOS, se obtiene la siguiente expresión para el tiempo de bajada:

$$\tau_f \approx \frac{C_L V_{DD}}{K_n (V_{DD} - |V_{Tn}|)^2} \tag{4.9}$$

Las expresiones calculadas para los tiempos de subida y bajada muestran que cambios en la tensión umbral (V_T) pueden causar importantes alteraciones en los retardos de conmutación (obsérvese la dependencia cuadrática). Por ejemplo, si V_T aumenta, los retardos de conmutación también se incrementan. El modelo a nivel lógico para este mecanismo es, evidentemente, ***delay***.

Otro indicador de la velocidad de conmutación viene dado por la respuesta en frecuencia (ω_0):

$$\omega_0 = \frac{g_m}{C_g} = \frac{2\mu(V_{GS} - V_T)}{L^2} \tag{4.10}$$

donde C_g es la capacidad puerta/canal.

Esta relación muestra que la velocidad de conmutación depende directamente de V_T y de la movilidad de los portadores, e inversamente del cuadrado de la longitud del canal. Un circuito rápido requiere un valor de g_m lo más alto posible. Cambios en V_T (o por tanto en g_m) afectan a ω_0.

4.3.1.1.3 Daños por plasma

Descripción

Debido a los nuevos elementos utilizados en los procesos de fabricación (técnicas, equipos, materiales, etc.), este mecanismo está teniendo mayor importancia [Shin *et al.* 1993]. En los procesos de fabricación que conllevan el empleo de plasma[38] (como el grabado, la deposición, la limpieza y la implantación iónica) se pueden dañar las finas capas de óxido de los dispositivos MOS. Cada vez con más frecuencia, en las tecnologías más avanzadas se están utilizando los denominados procesos "secos", que utilizan plasma, sustituyendo a los procesos "húmedos", en los que el tratamiento se realiza mediante productos químicos (básicamente ácidos).

El plasma incidente acumula carga en los electrodos de metal o polisilicio, habitualmente en las regiones donde el área o la periferia son grandes [Fang y McVittie 1993]. Dicha carga se transfiere a la región del correspondiente electrodo, donde un intenso campo eléctrico permite un cierto flujo de corriente a través de la fina capa de óxido de la puerta [McVittie 1996]. La carga acumulada puede ser positiva o negativa, y corresponde a iones móviles.

[38] Gel ionizado.

Efectos

El efecto más comúnmente observado cuando se da este tipo de mecanismo es un incremento en la corriente de fuga de la capa de óxido, a causa de la reducción de la tensión de rotura de la capa de óxido. En los dispositivos MOS también se observa una degradación de las características I_{DS}–V_{GS} [Rangan *et al.* 1999, Pagaduan *et al.* 2001], manifestándose generalmente como un incremento de la tensión umbral (V_T) [Lin *et al.* 1996].

Modelos

Los modelos asociados a este mecanismo de fallo pueden ser los mismos que se dedujeron al analizar el aumento de la corriente de fuga en la fase de desgaste del mecanismo de ruptura de la capa de óxido de la puerta (ver apartado 4.3.1.1.1): ***indetermination***, ***delay*** y ***bit-flip***.

En cuanto a la degradación de las características I_{DS}–V_{GS} y V_T, se pueden utilizar los mismos modelos que en el mecanismo de inyección de portadores "calientes": ***indetermination*** y ***delay***.

4.3.1.2 Metalización

De este grupo de mecanismos de fallo se van a destacar la electromigración, la migración inducida por estrés (en inglés *stress voiding*), así como algunos otros mecanismos con similares comportamientos y efectos.

4.3.1.2.1 Electromigración

Descripción

A medida que la tecnología avanza en la reducción de los tamaños de los circuitos submicrónicos y el aumento de la densidad de integración, los requisitos para mantener la inmunidad a la electromigración se acercan a su límite. Las cada vez más delgadas conexiones metálicas, junto con el aumento de la densidad de corriente han llevado el problema de la electromigración a ser uno de los referentes en la Fiabilidad de los circuitos integrados. Dichas consideraciones se justifican atendiendo a la expresión para el tiempo medio hasta la avería (en inglés *Mean Time to Failure*, MTTF) para la electromigración [Hawkins 2000]:

$$MTTF = \frac{A}{J^2} e^{Ea/kT} \tag{4.11}$$

donde A es el área de la sección de metal, J es la densidad de corriente, Ea es la energía de activación, k es la constante de Boltzman, y T es la temperatura absoluta.

Tanto en las tecnologías actuales como en futuras se utilizan entre 3 y 8 capas de metal, lo que se traduce en entre 50 y 500 millones de vías y contactos en circuitos integrados de entre 20 y 100 millones de transistores. Las vías actuales son pequeñas (inferiores a 0.5 μm) y frágiles. Con estas dimensiones, tienen el suficiente metal como para permitir la conducción, pero la electromigración puede deteriorarlas. Cuando a través de las finas líneas de metal se hacen pasar elevadas densidades de corriente, la fuerza ejercida por el flujo de electrones puede provocar el desplazamiento de los átomos de metal en el mismo sentido. Este movimiento de los átomos de metal se traduce en una acumulación[39] en el extremo positivo de la línea

[39] Denominada montículo (del inglés *hillock*).

metálica, y una ausencia[40] en el negativo. Para que el flujo de electrones provoque el desplazamiento de los átomos de metal, es preciso que se den unas condiciones especiales, como una distribución no homogénea en la línea metálica o un gradiente térmico.

La inclusión de pequeños porcentajes de cobre (hasta un 4%, y habitualmente alrededor de un 1%) en el aluminio ha permitido el aumento de la densidad de corriente (J) antes de que se produzca la electromigración. Este hecho se atribuye a la adsorción de cobre en los límites de los granos de aluminio. De esta forma, ocupan posiciones necesarias para el movimiento de los átomos de aluminio [Ghate 1981].

En la actualidad, el aluminio está siendo reemplazando a gran escala por cobre en la industria de semiconductores [Tammaro 2000, Ogawa *et al.* 2001]. El cobre se utiliza en circuitos integrados de altas prestaciones para reducir la capacidad intermetálica e incrementar la conectividad (debido a que su resistividad es entre un 30 y un 40% menor que la del aluminio). Si se considera, además, que la energía de activación para el cobre es superior que para el aluminio, lo que se traduce en una mayor resistencia a la electromigración, esta tendencia va a tener un impacto positivo en la reducción de la tasa de fallos permanentes.

Efectos

Eléctricamente, las averías pueden ocasionar un incremento de la resistencia de interconexión, lo que puede llevar incluso a un circuito abierto. La electromigración también puede producir cortocircuitos entre conexiones contiguas (en la misma capa) o situadas en capas adyacentes.

Modelos

A nivel electrónico, el efecto de la electromigración se puede modelar mediante cortocircuitos y circuitos abiertos en las conexiones metálicas. Éstos, pueden ocasionar a su vez diversos tipos de fallos a nivel lógico [Gil 1999]: ***stuck-at***, ***short***, ***open-line*** (en ocasiones ***stuck-open***), ***bridging***, ***indetermination*** y ***delay*** (véase la Figura 4.2).

4.3.1.2.2 Migración inducida por estrés (*stress voiding*)

Descripción

Durante el proceso de fabricación de la oblea, el encogimiento posterior a la deposición de las diferentes capas puede provocar un fuerte estrés mecánico en las capas individuales [Jones 1987]. Las finas películas metálicas son particularmente sensibles a dicho estrés, por lo que es posible que se produzca una transferencia de átomos desde las zonas que soportan un fuerte estrés para igualar las tensiones. El resultado de esta migración de átomos es una deformación de las capas de metal. En los dispositivos submicrónicos, el estrés por tensiones está asociado con las diferencias en el coeficiente de expansión térmica entre el metal y las películas de pasivación [Tezaki *et al.* 1990].

Efectos

Las manifestaciones físicas de este mecanismo de fallo son, por un lado, la formación de huecos (en inglés *voiding*) y de muescas (en inglés *notching*) en las líneas metálicas [Oates 1993], que producen un aumento de la resistencia y, en ocasiones, circuitos abiertos. Por otro, también se ha observado la formación de "rebarbas" metálicas (en inglés *whiskers*) [Turner y Parsons 1982], que pueden producir cortocircuitos.

---

[40] Conocida como hueco, *void* en inglés.

Modelos

Como ocurría con la electromigración, los fallos a nivel físico son circuitos abiertos o cortocircuitos en las conexiones metálicas, por lo que los modelos propuestos a nivel lógico son básicamente los mismos que en aquel caso: ***stuck-at***, ***short***, ***open-line***, ***stuck-open***, ***bridging***, ***indetermination*** y ***delay***.

4.3.1.2.3 Otros

Además de los ya vistos, existen otros mecanismos asociados a las metalizaciones cuyos efectos a nivel lógico se manifiestan de forma muy similar. Se pueden destacar la migración en los contactos, la migración en las vías, microfracturas, y los defectos de cobertura de escalones (en inglés *step coverage*).

A continuación se describen los diferentes mecanismos y sus efectos, y por último se especifican de forma conjunta los modelos de fallos equivalentes.

Mecanismos y efectos

- **Migración en los contactos (*contact migration*)**:

  Este mecanismo consiste en, bajo determinadas condiciones de estrés (de corriente o temperatura), la interdifusión (migración) de los átomos de metal y silicio en los contactos metálicos. La migración se da en los contactos de aluminio en silicio, y en función del tipo de estrés se puede dar en ambos sentidos: del aluminio hacia el silicio y viceversa.

  La unión física entre el metal y el semiconductor se realiza, de manera ideal, mediante el intercambio de unos pocos átomos de cada elemento. Sin embargo, si el intercambio no se controla, la interdifusión puede continuar después del proceso de fabricación, y el silicio desaparece de la superficie del contacto.

  La migración del silicio hacia el metal provoca un hueco en el contacto (en inglés *pitting*) originando un aumento de la resistencia de contacto e incluso un circuito abierto. Si la migración se produce del metal al silicio, se forman puntas o hendiduras de metal (en inglés *spiking*) en la zona de difusión por las que aumentan las corrientes de fuga en las uniones.

- **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. La diferencia principal con la migración en los contactos consiste en que el vaciado puede ocurrir en ambos sentidos del flujo de electrones [Oates 1994]. La existencia de oxígeno residual en las vías favorece este tipo de fallos [McPherson 1994].

  Se han observado incrementos resistivos y circuitos abiertos, dependiendo de la dirección de del flujo de electrones y del tipo de metal utilizado [Oates 1994].

- **Microfracturas y defectos de cobertura de escalones**:

  La topografía de la superficie de una oblea puede contener regiones con escalones pronunciados, tal como se muestra en la Figura 4.6. La capa de metal depositada en el escalón es muy fina, lo que incrementa su susceptibilidad a la electromigración por el pa-

so de elevadas densidades de corriente [van der Pol *et al.* 1996], 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 [Saito *et al.* 1993], pues la migración en los contactos se ve favorecida.

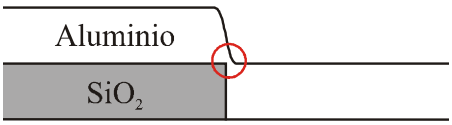

Figura 4.6: Ejemplo de escalón con riesgo de microfractura [Amerasekera y Najm 1997].

Pueden producirse incrementos en la resistencia de interconexión, circuitos abiertos o mecanismos relacionados con la electromigración.

Modelos

Puesto que los efectos de todos estos mecanismos se pueden resumir como circuitos abiertos y cortocircuitos en las interconexiones metálicas [Amerasekera y Najm 1997], 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***.

4.3.1.3 Encapsulamiento y ensamblado

El creciente tamaño de los chips requiere la utilización de mayores encapsulados y complejas técnicas de encapsulamiento, lo cual supone un aspecto importante en la consecución de la fiabilidad de los circuitos integrados. La complejidad del encapsulado puede constituir uno de los principales limitadores del rendimiento (en inglés *yield*) en la producción masiva de circuitos integrados VLSI (microprocesadores y circuitos programables).

En la Figura 4.7 se representa la sección de un típico encapsulado [Amerasekera y Najm 1997]. El **soporte de los pines** (de plomo) lo componen el conjunto de los pines; el chip se sujeta al **soporte del chip** (también de plomo), y se conecta con los pines mediante finísimos cables de conexión (habitualmente de oro).

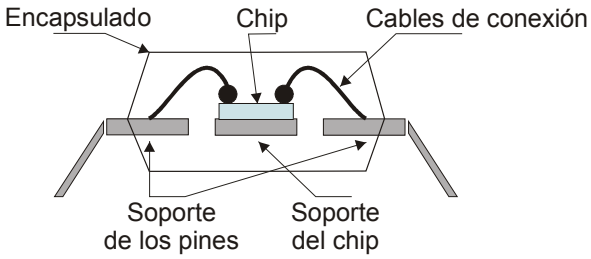

Figura 4.7: Sección de un encapsulado.

Se pueden identificar cuatro tipos de mecanismos de fallo:

1. Defectos de fijación del chip al soporte del chip (en inglés *die attach*).
2. Defectos de conexión en los cables que unen el chip con los pines.
3. Deslaminación entre el encapsulado plástico y el chip.

4. Mecanismos de fallo relacionados con la humedad, como la corrosión, la deslaminación y el "efecto palomita de maíz" (en inglés *popcorn effect*).

En los siguientes apartados se describen los efectos de algunos de los mecanismos de fallo particulares. Finalmente, se proponen para todos ellos una serie de modelos de fallos a nivel lógico.

4.3.1.3.1 Defectos de fijación del chip (*die attach*)

Descripción

La fijación del chip al soporte se realiza mediante calentamiento, utilizando elementos adhesivos especiales (compuestos por mezclas de metales –oro, plata, níquel, hierro, plomo, estaño, etc.– y otros materiales como silicio, carbono, etc.). Los defectos de fijación pueden ocasionar dos tipos de mecanismos de fallo. Por un lado, la aparición de huecos (*voids*) en la capa de adhesivo, que en procesos térmicos puede provocar la rotura (en inglés *crack*) del chip o del soporte. Además, la presencia de huecos puede originar zonas localizadas de alta disipación térmica. El segundo mecanismo está originado por 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.

Efectos

Los efectos de los mecanismos expuestos están relacionados con la fusión (en inglés *burnout*) del chip, variaciones paramétricas o la corrosión. Además, se puede producir la rotura del chip durante procesos con excesivo estrés mecánico o térmico. Durante el ciclo térmico y en el test de estrés acelerado (en inglés *High Accelerated Stress Testing*, HAST) se detectan muchos fallos de este tipo.

4.3.1.3.2 Defectos de conexión

Descripción

La conexión del chip con los pines (ver Figura 4.7) se realiza mediante un cable y dos conexiones: una con el chip y otra con el soporte de los pines. El material más comúnmente utilizado para los cables es el oro, por su resistencia a la corrosión y por su facilidad de fabricación, aunque a veces también se utiliza aluminio. Las conexiones se realizan entre el cable y sendas metalizaciones en el chip y en el soporte. En la Figura 4.8 se muestra un detalle de una conexión.

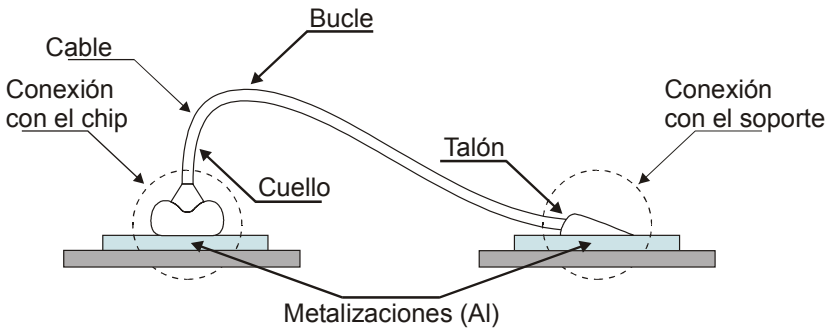

Figura 4.8: Detalle de una conexión [Amerasekera y Naijm 1997].

Las conexiones entre el cable y una metalización se pueden realizar de dos maneras: en forma de bola y en forma de cuña, cada una de las cuales se forma mediante técnicas diferentes. En la figura, la conexión con el chip es en forma de bola y la realizada con el marco en forma de cuña.

En la figura se resaltan algunos puntos débiles de la conexión:

- El cuello y el talón. Si el cable sufre tensiones (mecánicas) fuertes, la unión con el anclaje puede romperse por ellos, provocando circuitos abiertos.
- El bucle también es sensible a las tensiones del cable. Si es demasiado estrecho, tensiones fuertes pueden provocar la fractura del cable. Si, por el contrario, la tensión es demasiado débil, el cable puede moverse, provocando cortocircuitos con cables adyacentes.
- Al soldar los anclajes a las metalizaciones se producen aleaciones entre el oro del cable y el aluminio de la metalización. 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.

Efectos

El fallo más comúnmente observado son los circuitos abiertos debidos a desplazamientos en las conexiones con el chip. La formación de defectos en las aleaciones puede ocasionar altas resistencias en las conexiones de los cables [Shirley y Blish 1987]. El desplazamiento de los cables (en inglés *wire sweeping*) ocasionado por la inyección de plástico en el molde puede resultar en cortocircuitos entre cables adyacentes. La aparición de "rebarbas" (*whiskers*) también puede provocar cortocircuitos entre cables adyacentes. Si los cables están sometidos a fuertes tensiones mecánicas en las conexiones, pueden llegar a romperse, ocasionando circuitos abiertos. El adelgazamiento de los cables, que se produce especialmente cuando se utiliza aluminio, puede ocasionar calentamiento en zonas del cable de alta resistencia. El adelgazamiento se debe a la oxidación del cable de aluminio, que reduce la sección efectiva. Elevadas densidades de corriente pueden llegar a producir circuitos abiertos [King *et al.* 1989].

4.3.1.3.3 Deslaminación y "efecto palomita de maíz" (*popcorn effect*)

Descripción

Estos mecanismos están relacionados con la humedad. 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). El segundo se conoce como "efecto palomita de maíz" (en inglés *popcorn effect*), a causa del característico ruido que hace el chip al romperse.

Efectos

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.

4.3.1.3.4 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.

4.3.1.3.5 Modelos asociados

Los cuatro mecanismos descritos anteriormente se manifiestan a largo plazo como circuitos abiertos o cortocircuitos en las líneas. Por lo tanto, los modelos propuestos son básicamente los mismos que para la electromigración. Estos fallos pueden ser permanentes, o intermitentes mientras las conexiones son inestables y el defecto aún no se ha vuelto irreversible.

4.3.2 Fallos intermitentes

Acerca de este tipo de fallos se pueden hacer las siguientes consideraciones [Constantinescu 2001, Constantinescu 2002, Shivakumar *et al.* 2002]:

- Son fallos que ocurren repetidamente en el mismo circuito.
- La reparación del circuito afectado elimina el fallo.
- Los fallos intermitentes generados por variaciones en el proceso de fabricación y por residuos derivados de dicho proceso van a representar una importante fuente de errores en los actuales y futuros circuitos integrados submicrónicos.
- Tiene mayores tasas de fallo[41] que los fallos transitorios.
- A medida que las geometrías se reducen, algunos fallos permanentes se manifiestan inicialmente como intermitentes. Por ejemplo:
  - ⇨ Circuitos abiertos en las secciones más estrechas de los cables a causa de la electromigración.
  - ⇨ Cortocircuitos entre conductores adyacentes, en las zonas donde la capa de dieléctrico es más delgada.
  - ⇨ Corrientes de fuga en las capas de óxido debidas al efecto túnel directo.
- Las elevadas frecuencias de funcionamiento aumentan el impacto de las indeterminaciones temporales, que provocan violaciones de los márgenes temporales de seguridad ($t_{set\text{-}up}$, t_{hold}, etc.). Como consecuencia, se observa un aumento de la tasa de fallos intermitentes debida a violaciones de los márgenes de tiempo.

Los modelos de fallos deben ser similares a los de los fallos permanentes. De hecho, los mecanismos de desgaste se manifiestan inicialmente como fallos intermitentes hasta que el fallo se vuelve permanente. Por ello, 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).

[41] Número de fallos por unidad de tiempo.

4.3.3 Fallos transitorios

Los fallos transitorios, también llamados en inglés *soft errors* y *single event upsets* (SEU), son fallos temporales que puede surgir durante la operación de un circuito por diferentes causas, tanto internas como externas. A diferencia de los fallos permanentes (y la mayor parte de los intermitentes), los transitorios no introducen defectos físicos en el circuito, y tienen lugar durante un intervalo de tiempo relativamente corto.

Su efecto hace que el comportamiento del circuito afectado varíe de forma imprevisible, produciendo a menudo resultados incorrectos. El tratamiento de estos fallos es muy problemático debido a su ilocalización espacial y a su corta duración, por lo que las técnicas de test de fallos permanentes no se pueden aplicar en los transitorios. Si, además, se tiene en cuenta su importante incidencia en los sistemas digitales (se considera que hasta el 85% de las averías están producidas por fallos transitorios [Iyer y Rosseti 1986]), el desarrollo de sistemas tolerantes a fallos transitorios es un asunto complejo.

Tradicionalmente, la radiación (y en particular la debida a las partículas α y a los rayos cósmicos) ha sido considerada como la causa principal de fallos transitorios en un sistema. Además, debido a su corta duración, y sobre todo a los mecanismos naturales de enmascaramiento de que disponen los circuitos combinacionales, su efecto se restringía a los elementos de memoria. El modelo utilizado es el ***bit-flip***.

Los mecanismos naturales de enmascaramiento de los circuitos combinacionales son [Lidén *et al.* 2002, Shivakumar *et al.* 2002, Constantinescu 2002]:

- Enmascaramiento lógico. Este fenómeno ocurre cuando se produce un fallo en una parte del circuito cuya salida no afecta a la salida del sistema (porque no está activa o porque la salida depende de otras partes del circuito).
- Enmascaramiento eléctrico. Consiste en la atenuación de un pulso erróneo tras atravesar una serie de puertas en etapas posteriores, hasta el punto de que el pulso no afecte a la salida.
- Enmascaramiento producido por la ventana temporal de captura[42] de los biestables. Este mecanismo está relacionado con la captura de un fallo en un biestable, convirtiéndose en un error. La Figura 4.9 muestra cómo funciona este mecanismo.

  La ventana temporal de captura es el tiempo entre $t_{set\text{-}up}$ y t_{hold}. Si un pulso erróneo comienza antes de $t_{set\text{-}up}$ y finaliza después de t_{hold}, será capturado, provocando por tanto un error (de tipo *bit-flip*). Si termina antes de $t_{set\text{-}up}$ o empieza después de t_{hold}, el fallo será enmascarado. En caso de que el pulso solape parcialmente la ventana, pueden darse distintas situaciones. El pulso puede ser enmascarado (así se considera en [Shivakumar *et al.* 2002]) o bien puede ocasionar un estado metaestable transitorio [Texas Instruments 1997]. Este estado metaestable se traduce en una indeterminación transitoria del estado del biestable, seguida de un posible error no previsible cuando el biestable fluctúe aleatoriamente a uno de los dos estados estables.

Se da la paradoja de que algunos de los aspectos que más han influido en la reducción de las geometrías de los nuevos transistores submicrónicos son también la causa (directa o indi-

[42] En inglés, *latching-window masking*.

recta) del aumento de la tasa de fallos transitorios. Se pueden destacar, por ejemplo, la disminución de la tensión de alimentación y el aumento de la velocidad de conmutación (y en particular, de la frecuencia de funcionamiento).

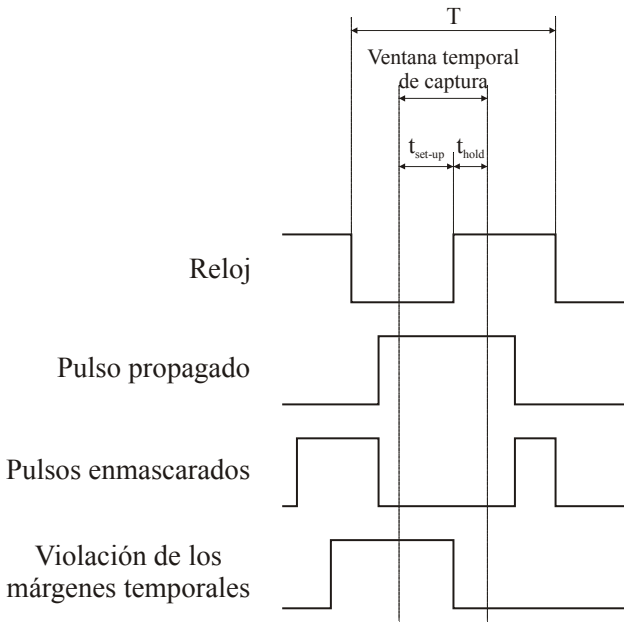

Figura 4.9: Enmascaramiento por ventana temporal de captura [Shivakumar *et al.* 2002].

Para que un transistor conmute su valor, es preciso superar una barrera de energía, representada por la denominada carga crítica (Q_{crit}). Es el valor mínimo de carga necesario para hacer conmutar a un nodo (transistor, celda de memoria, etc.), y es diferente para cada tecnología. Uno de los aspectos que más influyen en Q_{crit} es la tensión de alimentación. A medida que la tensión de alimentación de las diferentes tecnologías submicrónicas disminuye, Q_{crit} también lo hace. En [Hazucha y Svennson 2000] se propone la siguiente expresión para el cálculo de Q_{crit} para una celda SRAM (ecuación (4.12)):

$$Q_{crit}(V_{CC}, T) = C_0 \left(V_{CC} + (V_{CC} - V_{CC0}) \frac{T}{T_0} \right) \tag{4.12}$$

En la expresión, C_0 representa la capacidad efectiva del nodo, V_{CC} es la tensión de alimentación, V_{CC0} está relacionada con la tensión umbral de la tecnología, T es la constante de tiempo de carga y T_0 normaliza la dependencia respecto a la constante de tiempo. Las tres constantes (C_0, V_{CC0} y T_0) son dependientes de la tecnología, en particular de la longitud de la puerta (L_G). Así, al simular con SPICE la ecuación (4.12) para diferentes tecnologías (0.8 µm, 0.6 µm, 0.35 µm y 0.1 µm), observaron que, al pasar de la tecnología de 0.6 µm a la de 0.1 µm, la carga crítica disminuye casi cuadráticamente con L_G.

En [Tosaka *et al.* 1999] se utiliza una expresión general para cualquier dispositivo, considerando únicamente la relación lineal con la tensión de alimentación y la capacidad del nodo (ecuación (4.13)):

$$Q_{crit} = V_{DD} C_{node} \tag{4.13}$$

De la expresión en la ecuación (4.13) se deduce que Q_{crit} depende del factor de escalado de forma cuadrática ya que tanto la tensión como la capacidad dependen linealmente del escalado [Anelli 2000].

Sin embargo, la carga crítica no depende únicamente de la capacidad del nodo y de la tensión de alimentación. Factores como el tipo de transistor (NMOS, PMOS o bipolar), la unión afectada (colector-emisor o colector-base en transistores bipolares), el estado inicial del transistor (a '0' o a '1'), la tecnología de fabricación (Bulk o SOI), etc. también influyen en la carga crítica. En [Chee y Tsang 2002, Srinivasan *et al.* 1994, Freeman 1996, Hazucha y Svensson 2000, Gautier y Leroy 2001, Holbert 2003] se pueden ver cálculos para la carga crítica considerando dichos aspectos.

		0.8 µm		0.6 µm		0.35 µm		0.1 µm	
		5 V	3.3 V	5 V	3.3 V	3.3 V	2.5 V	1.2 V	0.9 V
Tipo de transistor	PMOS	300	200	210	130	105	80	12	9
	NMOS	200	100	200	110	80	50	7	5

Tabla 4.1: Valores de la carga crítica (en fC) para diferentes tecnologías CMOS, distinguiendo entre los valores de tensión de alimentación para altas prestaciones y bajo consumo y en función del tipo de transistor.

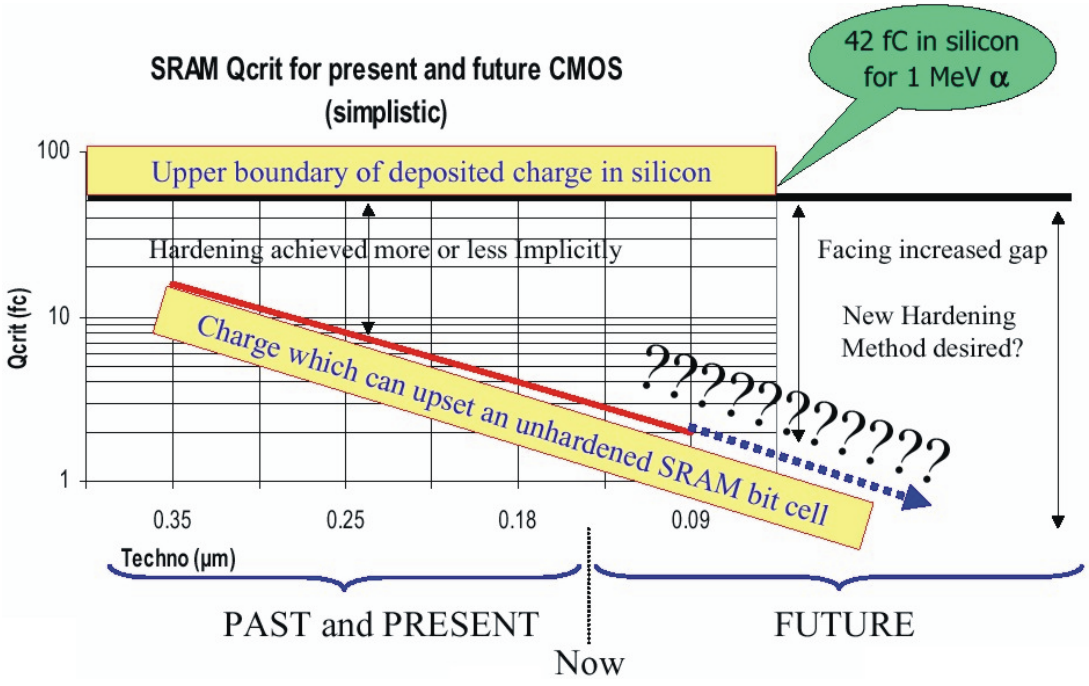

Figura 4.10: Tendencia de la carga crítica en el futuro [Gautier y Leray 2001].

En la Tabla 4.1 (deducida de [Hazucha y Svensson 2000]) se muestran los valores de Q_{crit} para diferentes tecnologías y tensiones de alimentación (para cada tecnología se consideran dos versiones con tensiones de alimentación diferentes; la más alta es para altas prestaciones, y la baja para bajo consumo).

La Figura 4.10, presentada en el RADECS 2001[43] pronostica la tendencia de dicho parámetro en el futuro. Como se puede comprobar, cada vez es necesaria una menor cantidad de energía para hacer conmutar a los transistores. Por ello, dado que existen mecanismos, tanto externos a los circuitos como internos, que pueden generar energía, cada vez les resultará más fácil alterar el valor de los transistores de un circuito.

El aumento de la velocidad de conmutación afecta a los mecanismos de enmascaramiento de los circuitos, y más en particular, al enmascaramiento eléctrico. Como por otro lado también se observa una reducción de la profundidad de las series de puertas en las diferentes etapas en sistemas segmentados (en inglés *pipelined*) [Shivakumar *et al.* 2002], el efecto de atenuación (y eliminación) eléctrica de los pulsos desaparece.

En los microprocesadores actuales, además, se han incrementado notablemente el número de etapas de segmentación (en inglés *pipelining*), lo que ha permitido el aumento de la frecuencia de funcionamiento. Esto ha ocasionado que los márgenes temporales se reduzcan. Como resultado, el enmascaramiento relacionado con la ventana temporal de captura se ha reducido considerablemente, y es más fácil que un pulso erróneo se transmita a un biestable [Shivakumar *et al.* 2002].

Como se puede ver, no sólo se producen más fallos transitorios, sino que además se producen en lugares que tradicionalmente no se consideraban en los modelos de fallos: los circuitos combinacionales. Estos fallos además, se pueden propagar de forma más sencilla al resto del sistema, bien a través de su captura en biestables o celdas de memoria, o por otros circuitos combinacionales, dando lugar a múltiples errores ocasionados por un único fallo.

Un aspecto muy importante de los fallos transitorios es su duración. En realidad, este es un tema abierto, ya que no hay una definición clara. Se debe a que en la duración de un fallo transitorio influyen muchos factores. De hecho, son los mismos de los que depende la carga crítica: tipo de tecnología (bipolar/CMOS/...), el proceso de fabricación (SOI/Bulk/...), el tipo de transistor en el que se produce el fallo, el mecanismo de fallo, etc. Por este motivo, en la bibliografía sólo hay aproximaciones en función del caso concreto que se estudia.

En los próximos apartados se describen los dos mecanismos de fallo relacionados con la radiación que afectan en mayor medida a los circuitos VLSI (las debidas a las partículas α y a los rayos cósmicos), explicando sus efectos y asociándoles una serie de modelos de fallo en los niveles lógico y RT. Finalmente se comentan otros mecanismos de fallo relacionados directamente con las nuevas tecnologías de fabricación de circuitos VLSI.

4.3.3.1 Radiación de partículas α

En los materiales utilizados en los encapsulados de los circuitos integrados existen impurezas radiactivas como torio, uranio, etc. Dichos elementos radiactivos son capaces de emitir partículas α con alta energía (hasta 8 MeV). Cuando la radiación alcanza al silicio, se generan pares electrón-hueco que pueden producir cargas de alrededor de 40 fC [Borel 2001]. Si el número de electrones en movimiento es lo suficientemente elevado como para sobrepasar la carga crítica (Q_{crit}) del transistor afectado, su comportamiento se verá alterado.

[43] RADECS son las siglas de la *European Conference on Radiation and its Effects on Components and Systems*. La edición del año 2001 fue la 6ª.

Tradicionalmente, este mecanismo de fallo se ha asociado a las celdas DRAM y a los registros dinámicos. Por ejemplo, cuando una celda DRAM con un '1' almacenado se ve afectada por la radiación α, cambia su valor a '0'; sin embargo, si el valor almacenado es un '0', no ocurre nada (en realidad, lo que sucede es que no se produce un incremento apreciable de carga) [Amerasekera y Najm 1997]. Este efecto se puede modelar con un ***bit-flip*** "selectivo".

Sin embargo, este modelo no es válido cuando el fallo se produce en un circuito combinacional, porque el efecto del fallo es distinto: cuando el efecto del impacto desaparece, el transistor (y, por consiguiente, el circuito al que pertenece) recupera el estado correcto. El motivo es 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 el transistor puede producir una salida indeterminada (modelo ***indetermination***) [Gil 1999].

En [Juhnke y Klar 1995] se calcula la tasa de fallo (en inglés *soft error rate*, SER) debido a partículas α, incluyendo tecnologías futuras. Sus resultados apuntan a que, para una determinada tecnología (especificada por su carga crítica), a medida que disminuye la tensión de alimentación, la tasa de fallo aumenta en gran manera. La causa principal es que la reducción de la tensión de alimentación implica un descenso acusado de la carga crítica lo que facilita la superación de ésta por parte de la partículas α.

Por ejemplo, en el caso de la tecnología de 0.6 μm, reducir la tensión de alimentación a la mitad de su valor normal supone multiplicar la tasa de fallo por 8. Si se consideran conjuntamente los dos parámetros de escalado (el tamaño característico[44] del proceso de fabricación y la tensión de alimentación), se puede esperar una mayor sensibilidad a las partículas α en las nuevas tecnologías CMOS, aumentando su influencia en las futuras generaciones de semiconductores.

Por lo que respecta a la duración de los fallos ocasionados, en [Srinivasan *et al.* 1994] se modela la onda de la corriente generada por el impacto de una partícula α en transistores bipolares con la ecuación (4.14):

$$I(t) = \frac{Q_{total}}{\tau_f - \tau_r}\left(e^{-t/\tau_f} - e^{-t/\tau_r}\right) \qquad (4.14)$$

donde Q_{total} es la carga total generada, y τ_r y τ_f son respectivamente las constantes de tiempo de subida y bajada (dependientes de la tecnología). A partir de esta ecuación, y usando un simulador electrónico, es posible generar una librería (tabla) de duraciones de pulso para diferentes tecnologías (fijando τ_r y τ_f), variando el valor de Q_{total}.

Con los valores de las constantes de tiempo aplicadas se obtienen unos pulsos de corriente de duración inferior a 1 ns. Posteriormente, estos pulsos de corriente se simulan en el circuito y se obtienen pulsos de tensión. En el trabajo no se da ninguna información sobre su duración a nivel lógico.

---

[44] Del inglés *feature size*.

En [Cha *et al.* 1993] se simula con SPICE3 una expresión similar, aplicada a un circuito CMOS en tecnología de 2 μm. Introduciendo valores de carga entre 1 pC y 9 pC, y considerando una tensión umbral de 2.5 V, se obtuvieron a nivel lógico pulsos de tensión cuya duración oscilaba entre 0.6 ns y 1.6 ns.

En [Singh y Koren 2003] se aplica la misma fórmula a un conversor A/D, y suponiendo un rango de carga generada de entre 1 pC y 5 pC, se obtienen pulsos de corriente de alrededor de 5 ns de duración.

4.3.3.2 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.

El efecto del impacto de un neutrón en el silicio es similar al de una partícula α, sólo que en este caso puede liberar una carga de cientos de fC en una pocas micras (frente a los aproximadamente 10 fC/μm [Amerasekera y Najm 1997] que libera una partícula α), que puede superar más fácilmente (y con mucha mayor probabilidad) la carga crítica de los transistores (Q_{crit}). Por este motivo, su efecto es más nocivo.

Cuando un neutrón impacta con el silicio, la carga de los transistores puede verse seriamente afectada, de tal modo que (si la carga generada es superior a la carga crítica, Q_{crit}) su estado puede cambiar, haciendo conmutar el valor lógico del circuito al que pertenece el transistor. Este valor erróneo, cuando el fallo se produce en un elemento de almacenamiento (registros y memorias), permanece hasta que se sobrescribe. Por este motivo, el modelo más frecuentemente utilizado para representar los efectos de este mecanismo de fallo es ***bit-flip***. Como ya se comentó en el apartado anterior, también hay que considerar el modelo ***pulse*** para elementos combinacionales.

Al igual que en la radiación de partículas α, también hay que considerar el modelo ***indetermination***.

Tradicionalmente, la tasa de fallo (SER) debida a los rayos cósmicos se ha calculado despreciando el efecto de los neutrones de baja energía (menor de 1 MeV). Sin embargo, por la disminución de la carga crítica en las tecnologías actuales, neutrones de baja energía (con mayor flujo que los de alta energía [Ziegler 1998]) también son capaces de provocar fallos, incrementando notablemente la tasa de fallo. Además, dado el cada vez más reducido tamaño de los modernos (y futuros) transistores, también se detecta un aumento de la probabilidad de que el impacto de un neutrón de alta energía ocasione **fallos múltiples** [Constantinescu 2002].

Al igual que ocurre con otros fallos transitorios la duración de los causados por partículas cósmicas es un asunto muy poco tratado. En [Freeman 1996] se modela la onda de la corriente generada por un impacto con la ecuación (4.15):

$$I(t) \propto \frac{Q}{T} \times \sqrt{\frac{t}{T}} \times e^{-t/T} \qquad (4.15)$$

donde Q es la carga almacenada debido al impacto de las partículas, y T es la constante de tiempo de carga del transistor para la tecnología bajo estudio (dependiente de la tecnología).

Como ya se explicó en el caso de las partículas α, se puede generar una librería de duraciones de pulsos para diferentes tecnologías (fijando T), simulando la ecuación (en un simulador electrónico) para diferentes valores de Q.

Las duraciones de los pulsos de corriente obtenidas en el trabajo oscilan entre los 200 ps y los 500 ps.

En [Mavis y Eaton 2000] se comparan los pulsos de corriente generados en las tecnologías Bulk CMOS y EEPROM. Mientras las duraciones en la primera oscilan entre 100 ps y 200 ps, en la segunda lo hacen entre 1 ns y 2 ns.

Lamentablemente, en todos estos estudios no se da ninguna información concreta acerca de las duraciones de los pulsos de tensión a nivel lógico. Sin embargo, dan una idea de la posible metodología a utilizar: inyectar pulsos de corriente en un circuito, simular a nivel electrónico, y establecer umbrales para generar los pulsos de tensión a nivel lógico.

4.3.3.3 Otros mecanismos

Además de la radiación, debido a las características de las interconexiones en los dispositivos submicrónicos (utilización de múltiples capas, reducción de los espesores de las líneas de conexión, disminución de la separación entre líneas (tanto adyacentes como en capas contiguas, etc.), hay que tener cada vez más en cuenta otras causas de fallos cuando funcionan a frecuencias elevadas. Básicamente se deben al incremento de la resistencia de las conexiones metálicas y de la capacidad parásita entre líneas de conexión. Esto afecta al retardo de propagación de las señales. Por ejemplo, los efectos piel (en inglés *skin effect*) y Miller pueden producir violaciones de los márgenes de tiempo ($t_{set\text{-}up}$, t_{hold}, etc.), al modificarse los retardos de los circuitos lógicos. 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.

El efecto piel [Walker 2000] es la concentración de la corriente eléctrica en la superficie de los cables de interconexión. Este efecto hace que la resistencia de la interconexión varíe con la frecuencia (a mayor frecuencia mayor resistencia).

El efecto Miller se produce cuando dos cables (pistas) adyacentes conmutan simultáneamente en direcciones opuestas. Cuando sucede esto, la capacidad efectiva entre terminales se modifica [Sylvester y Keutzer 1999].

Es importante destacar la alta probabilidad de que los fallos ocasionados por estos mecanismos sean múltiples, sobre todo de tipo *delay*. Por ejemplo:

- Cuando dos líneas resultan afectadas por el efecto Miller, se modifican los retardos de propagación de ambas. En este caso, la **multiplicidad** se produce **en el espacio**.
- Cuando el efecto piel afecta a una línea serie, todos los bits transmitidos a partir del momento del fallo resultarán alterados. Entonces se habla de fallos **múltiples en el tiempo**.

Otros mecanismos que pueden provocar fallos transtorios, 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 [Gil 1999]. Estos mecanismos también pueden originar fallos de tipo ***bit-flip***, ***pulse***, ***delay*** e ***indetermination***.

4.4 Resumen y conclusiones

En este capítulo se ha presentado un estudio de los principales mecanismos de fallo en las tecnologías submicrónicas actuales. De este estudio se han derivado un amplio conjunto de modelos (a los niveles lógico y de transferencia de registro, o RT) de fallos permanentes, intermitentes y transitorios, muy superior a los utilizados tradicionalmente: *stuck-at* para los fallos permanentes e intermitentes y *bit-flip* para los transitorios.

No sólo se han estudiado los mecanismos comunes de fallo (tanto internos como externos) de los circuitos integrados, sino también los efectos que las nuevas tecnologías submicrónicas (en cuanto a los procesos de fabricación, los materiales utilizados y la reducción de las geometrías) inducen en dichos mecanismos, así como en la aparición de otros nuevos.

La reducción de las geometrías ha provocado que algunos mecanismos de fallo se vean potenciados por las pequeñas dimensiones de los elementos que componen los circuitos digitales. Las capas de óxido, pistas, metalizaciones, etc., se ven afectadas por mecanismos de desgaste, como la rotura de la capa de óxido, la electromigración y los defectos de encapsulamiento. Estos mecanismos ocasionan fallos permanentes e intermitentes que, a nivel electrónico, se manifiestan básicamente como cortocircuitos y circuitos abiertos, y que a nivel lógico se pueden modelar como fallos *stuck-at*, *open-line*, *stuck-open*, *indetermination*, *delay*, *short* y *bridging*. Sobre todo, se espera un gran aumento de la tasa de fallos intermitentes.

En cuanto a los fallos transitorios, la reducción de las geometrías y de las tensiones de alimentación inciden en los mecanismos de fallo en dos vertientes. Por un lado, han ocasionado un notable descenso de la energía necesaria para hacer conmutar los transistores (la carga crítica, Q_{crit}). Este hecho, 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 (eléctrico, lógico y por ventana temporal de captura). Por otra parte, a causa de las reducidas distancias entre las conexiones de los chips han aparecido efectos capacitivos, que también se han visto potenciados por las elevadas frecuencias, que afectan a la respuesta temporal de los circuitos.

La reducción de la carga crítica ha provocado que tanto la radiación interna (debida a las partículas α) como la externa (debida a los rayos cósmicos) sean capaces de afectar en mayor medida que en las tecnologías clásicas. Tanto es así, que mecanismos de fallo que en tecnologías tradicionales tenían muy bajo impacto, como la radiación por partículas α y de rayos cósmicos de baja energía (por debajo de 1 MeV), en las tecnologías actuales presentan unas tasas de fallo cada vez mayores. Además, a causa de la atenuación de los mecanismos de enmascaramiento, los fallos ocasionados en la lógica combinacional pueden propagarse a los elementos de almacenamiento.

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*, existen 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 en cuanto a su efecto en el circuito afectado, se diferencian en cómo responde el circuito después del fallo. 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.

En cuanto a las consecuencias de los diferentes efectos capacitivos y resistivos que se producen en las metalizaciones de las tecnologías submicrónicas (entre los que se pueden destacar el efecto piel y el efecto Miller), todas ellas se pueden expresar mediante el modelo *delay*, que representa la alteración de los retardos de propagación de los circuitos.

En algunos trabajos publicados recientemente se demuestra que los nuevos avances tecnológicos (en relación con los materiales y procesos de fabricación y diseño) han mejorado la calidad de los semiconductores, lo que se ha reflejado en un fuerte descenso de la tasa de fallos permanentes (a principios de los años 90), observándose una estabilización posterior [Constantinescu 2002]. Por otro lado, aspectos tan positivos como el escalado de la tensión de alimentación y de las geometrías harán a los dispositivos muy sensibles a los fallos intermitentes y transitorios, cuyas tasas se incrementarán notablemente en el futuro [Constantinescu 2001, Shivakumar *et al.* 2002].

En este sentido, en [Shivakumar *et al.* 2002] se cuantifica la evolución de la tasa de fallos transitorios (SER) de los diferentes circuitos de un microcomputador (memoria, registros, y lógica combinacional). Los resultados son muy significativos: la SER de los circuitos combinacionales sufrirá un enorme aumento debido a la disminución de los efectos de enmascaramiento, lo que hará que hacia el año 2011 se equipare con la de las memorias.

4.5 Trabajo futuro

Algunos de los aspectos del trabajo expuesto en este capítulo se pueden ampliar, entre los que cabe destacar dos. Por un lado, la utilización de herramientas de simulación electrónica para estudiar el efecto de los fallos transitorios en nuevas tecnologías, o el análisis de las características temporales de los fallos transitorios, en particular su duración.

Otro tema a desarrollar, de hecho ya en curso como se explicará en el capítulo 7, 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.

5 Técnicas de inyección de fallos mediante simulación de modelos en VHDL

5.1 Introducción

El lenguaje VHDL se ha convertido en uno de los más apropiados desde el punto de vista de la simulación de fallos, debido a su capacidad de integrar diversos niveles de abstracción. Las razones del extendido uso de VHDL se pueden resumir en:

- Es ampliamente utilizado en el diseño digital.
- Ofrece la posibilidad de describir el sistema a distintos niveles de abstracción [Dewey y Geus 1992, Aylor *et al.* 1990] (puerta, RT, chip, algorítmico, sistema, etc.)[45] gracias a la posibilidad de descripciones estructurales y comportamentales.
- Ofrece una buena capacidad para realizar modelos de los sistemas a alto nivel.
- Ofrece posibilidades para realizar tareas de test [Miczo 1990].

Para realizar la inyección 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 [Arlat 1992, Jenn y Arlat 1992, Rimén *et al.* 1992, Jenn *et al.* 1993a, Jenn *et al.* 1993b]. El primero 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 [Ohlsson *et al.* 1992, Jenn *et al.* 1994], sin tener que modificar el código VHDL del modelo. El segundo grupo se fundamenta en la modificación del código VHDL del modelo, mediante la inserción de componentes perturbadores (*saboteurs*) en arquitecturas estructurales [Boué *et al.* 1996, Boué *et al.* 1998, Amendola *et al.* 1996, Folkesson *et al.* 1998], o creando mutaciones (*mutants*) de componentes ya existentes [Ghosh y Chakraborty 1991, Armstrong *et al.* 1992].

La técnica basada en las órdenes del simulador es más fácil de implementar, aunque depende del simulador empleado, y normalmente el conjunto de modelos de fallos es más restringido. Por el contrario, los perturbadores y los mutantes, aunque son técnicas más complejas de llevar a la práctica, permiten inyectar un conjunto de fallos más amplio.

Además de estas dos técnicas, se proponen algunas alternativas basadas en la ampliación de la declaración de tipos y funciones de resolución del código en VHDL. Podrían englobarse en un tercer grupo denominado “otras técnicas”.

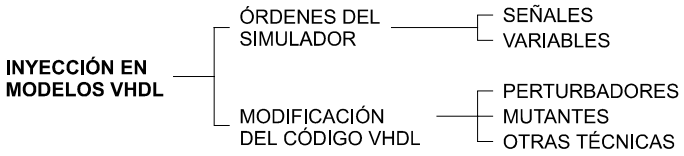

Figura 5.1: Clasificación de las técnicas de inyección sobre modelos VHDL [Baraza 1999].

[45] En el capítulo 4 ya se advirtió de la existencia de diferentes clasificaciones de los niveles de abstracción. El nivel de puerta utilizado en el presente capítulo coincide con el nivel lógico referido en el capítulo 4.

En la Figura 5.1 se muestran las diferentes técnicas esquematizadas.

El objetivo de este capítulo es, por un lado, profundizar en la descripción de estas técnicas. Además, como la herramienta de inyección de fallos desarrollada hace uso de ellas, se pretende mostrar los estudios realizados encaminados a la integración de estas técnicas en la herramienta VFIT.

La distribución del capítulo es la siguiente. En primer lugar, en el apartado 5.2 se hace un breve resumen del lenguaje VHDL, y se comentan algunas de las características especiales de VHDL que favorecen la inyección de fallos. En los apartados 5.3 y 1.1 se describen con detalle las técnicas de inyección expuestas Además se incluirán las propuestas realizadas para su integración en la herramienta VFIT. En el apartado 1.1 se realiza una comparación de las técnicas anteriormente descritas, y en el apartado 5.6 se presentan los trabajos más relevantes relacionados con las diferentes técnicas, incluyendo las herramientas de inyección de fallos existentes. Para finalizar, en el apartado se resumen las técnicas de inyección expuestas y se extraen las conclusiones más interesantes.

5.2 Características del lenguaje VHDL

Cuando a finales de los años 70 la complejidad de la realización de diseños digitales desbordaba las metodologías habituales, el gobierno de EE.UU. desarrolló el proyecto VHSIC[46] (1980) para la definición de un lenguaje de descripción de *hardware* (HDL, *Hardware Description Language*) con grandes capacidades:

- Permitir implementar diseños jerárquicos.
- Definir diferentes niveles de abstracción tanto en la interfaz como en la especificación del funcionamiento de los elementos.
- Permitir especificar el funcionamiento de los elementos de diferentes maneras.
- Manejar el tiempo tanto en descripciones de sistemas orientadas a su simulación como para sintetizarlo en un circuito.

El IEEE[47] plasmó la idea y surgió VHDL[48] con la norma IEEE Std 1076-1987 [IEEE 1988]. En 1993 se hizo una revisión y ampliación que se publicó con la referencia IEEE Std 1076-1993 [IEEE 1994].

A continuación se describen brevemente las características generales del lenguaje, así como alguno de los elementos de su semántica más útiles para la inyección de fallos.

5.2.1 Descripción general del lenguaje VHDL

Un modelo realizado en VHDL está formado por *componentes* (`components`) conectados por *señales* (`signals`). Estos componentes no son más que instancias de otros elementos de diseño más simples que se incorporan en el modelo más complejo.

VHDL permite que el valor de una señal pueda ser generado desde diferentes fuentes (denominadas *drivers*). Para ello, existe el concepto de **función de resolución**, cuya misión es

[46] *Very High Speed Integrated Circuits*.
[47] *Institute of Electrical and Electronic Engineers*.
[48] *VHSIC Hardware Description Language*.

generar el valor definitivo de la señal a partir de los valores de las distintas fuentes. Gracias a este elemento, es posible modelar *buses* y conexiones en colector abierto (obviamente, con funciones de resolución diferentes).

Independientemente de la complejidad del elemento modelado, un diseño en VHDL consta de una descripción de su interfaz y de su implementación.

La interfaz de un diseño se especifica en una unidad denominada entidad (`entity`), que contiene la especificación de sus entradas y salidas (puertos, o `ports`), así como de una serie de parámetros característicos inherentes al diseño (llamados constantes genéricas, o `generics`).

La implementación del diseño se lleva a cabo en una unidad llamada arquitectura (`architecture`), en la que se describe el funcionamiento del diseño. Esta descripción puede hacerse desde dos puntos de vista:

- Especificando los componentes (elementos más simples) que constituyen el diseño principal y cómo están conectados. Este tipo de descripción se denomina *estructural*.
- Mediante un modelo algorítmico que especifica el funcionamiento del diseño. Esta descripción se llama *comportamental*.

La arquitectura de un diseño se puede especificar de manera puramente estructural, puramente comportamental, o mediante una mezcla de ambas.

Puesto que su misión es describir circuitos digitales, una particularidad muy importante del lenguaje VHDL es que es **concurrente**. Es decir, sus sentencias se ejecutan en paralelo, salvo alguna excepción. Dicha excepción la constituyen las sentencias incluidas dentro de un proceso (`process`), que son la base de las descripciones comportamentales. Los procesos son unas sentencias especiales que, aún siendo concurrentes, están compuestas por otras sentencias que se ejecutan secuencialmente dentro del proceso. A diferencia del resto de sentencias, los procesos pueden comunicarse con otros procesos, a través de las señales de la arquitectura. Además, los procesos disponen de `variables`, exclusivas a cada proceso e inaccesibles desde el exterior, salvo en el caso particular de las variables compartidas[49] (`shared variables`).

Los procesos disponen de dos mecanismos que permiten la sincronización entre ellos:

- La lista de activación, que es una lista de señales cuya alteración "despierta" al proceso, que se encuentra "dormido" en espera de una modificación en alguna de señales de la lista.
- Las sentencias de espera (`wait`), que bloquean al proceso hasta que se cumpla alguna condición, que puede ser tanto booleana como temporal (especificando un valor de tiempo).

Las clases de sentencias que pueden aparecer dentro de un proceso son muy diferentes a las que se dan fuera de ellos. De hecho, las primeras se denominan *sentencias secuenciales*, y las segundas *concurrentes*. Sentencias concurrentes son los propios procesos, instancias de componentes, asignaciones de valor a señales, llamadas a procedimientos, etc. Entre las secuenciales cabe destacar las sentencias `if` y `case`, bucles y asignaciones de valor a variables y señales.

[49] También denominadas variables globales.

Los tipos de datos que se pueden manejar en un diseño en VHDL van desde los tipos simples definidos en las *bibliotecas* (`library`) más comunes, que manejan datos básicos como los valores lógicos en una línea o conjunto de líneas (tipos `bit`, `bit_vector`, `std_ulogic`, `std_ulogic_vector`, `std_logic`, `std_logic_vector`, etc.)[50,51], hasta estructuras compuestas que representen la información transmitida en modelos de alto nivel. Uno de los tipos de datos más útiles es el tipo `time`[50], que permite representar valores de tiempo, tanto para las esperas en las sentencias `wait` como para especificar retardos en las asignaciones de valor a las señales (mediante la opción `after`).

Cuando se inserta un componente en un diseño estructural, se debe indicar cuáles son las conexiones reales de los puertos de la instancia. Asimismo, también se puede configurar el valor de las constantes genéricas.

Una de las grandes ventajas de VHDL es el hecho de permitir diferentes implementaciones para un mismo sistema. Es decir, a una misma entidad se pueden asociar múltiples arquitecturas. Para poder simular un sistema en el que sus componentes pueden tener diferentes implementaciones (debidas a la existencia de arquitecturas distintas o a variaciones de las constantes genéricas de la entidad), existe el mecanismo de configuración (*configuration*), que permite elegir la implementación deseada para cada componente del diseño.

A continuación se describen algunos elementos del VHDL que son muy útiles para la inyección de fallos. En particular, se trata de los tipos lógicos "multivalor", las funciones de resolución, las constantes genéricas y el mecanismo de configuración.

5.2.2 Elementos del VHDL útiles para la inyección de fallos

5.2.2.1 Tipos lógicos "multivalor"

`Std_ulogic` y `std_logic` son tipos enumerados declarados en la biblioteca *IEEE*. Se utilizan para representar valores lógicos, y tienen un número elevado de valores diferentes:

'U': no iniciado
'X': indeterminación
'0': 0 lógico
'1': 1 lógico
'Z': alta impedancia
'W': indeterminación débil
'L': 0 débil
'H': 1 débil
'-': no importa (*don't care*)

porque no sólo representan el nivel lógico de tensión, sino también la fuerza con que ejercen el valor en la línea. Así, 'X', '0' y '1' son valores **fuertes**, mientras que 'W', 'L' y 'H' son **débiles**.

[50] Aunque los tipos de datos aquí expuestos no pertenecen realmente al lenguaje VHDL, puesto que no existe ningún tipo de datos predefinido en su semántica, sí pertenecen a algunas de las bibliotecas más comúnmente utilizadas en cualquier diseño: *STD* (especificada en la propia norma del lenguaje) e *IEEE*.

[51] Los tipos `std_ulogic`, y `std_logic`, etc. son tipos lógicos "multivalor" (traducción libre del inglés *multi-valued logic types* [Ashenden 2002]), en los que se incidirá en el apartado 5.2.2.1.

Estos nueve valores permiten representar bastante fielmente el comportamiento electrónico de diseños implementados con las tecnologías de fabricación de circuitos integrados más comunes: NMOS, CMOS y TTL.

5.2.2.2 Funciones de resolución

El modelado de sistemas en los que existen señales cuyo valor depende del de varias posibles fuentes interconectadas (como ocurre en los *buses* o en líneas en colector abierto) es, a priori, problemático. Dando por sentado que una señal sólo puede tener un valor en un instante de tiempo, hay que definir un mecanismo que determine, a partir de los valores de todas las fuentes en ese instante de tiempo, el valor que la señal tendrá de forma efectiva. La calificación "de resolución" se debe a que estas funciones resuelven los conflictos de valor en la señal.

Por este motivo, el lenguaje VHDL ofrece las funciones de resolución, muy íntimamente relacionadas con los tipos "multivalor" comentados anteriormente, para que el diseñador pueda especificar el comportamiento deseado en caso de conflictos en la línea.

De hecho, el tipo "multivalor" `std_logic` (declarado en la biblioteca *IEEE*, y explicado en el apartado anterior), no es más que el resultado de aplicar una función de resolución llamada `resolved` (y definida en la misma biblioteca), a un conjunto de señales del tipo `std_ulogic` (también visto con anterioridad), y que tiene los mismos valores lógicos que el original.

Junto con el tipo de datos generado a partir de la función de resolución la biblioteca incluye también la ampliación del comportamiento de las puertas lógicas para contemplar el uso del nuevo tipo de datos.

Esta función de resolución predefinida modela el comportamiento de las conexiones en tecnología CMOS. Para poder implementar otros tipos de conexiones o de tecnologías, el diseñador puede generarse su propia función de resolución para generar un nuevo tipo de datos, y deberá actualizar los modelos de las puertas lógicas para que se puedan aplicar al nuevo tipo de datos.

5.2.2.3 Constantes genéricas

Como ya se comentó en el apartado 5.2.1, estas constantes se declaran en la entidad del modelo, junto con los puertos.

Este tipo de constantes es muy útil, ya que permite realizar diseños en los que algún detalle dependa de un parámetro global. Estos "detalles" pueden ser elementos tan importantes como el tamaño de los buses y registros, retardos, etc.

Si, además, se tiene en cuenta que estos parámetros no son realmente constantes, dado que su valor se puede especificar tanto en el momento de insertar el componente en la arquitectura o mediante una configuración, este tipo especial de constantes adquiere cierta relevancia.

En particular, si se considera una constante genérica que represente un retardo, asignándole diferentes valores se puede modificar el comportamiento temporal del modelo.

Sin embargo, la importancia real de estas constantes en cuanto a su utilidad para la inyección de fallos reside en el hecho de que son modificables en tiempo de simulación utilizando órdenes del simulador, de un modo muy similar a como se hace con las variables[52], como se refiere en el apartado 5.3. Por este motivo, la inyección de fallos de tipo *delay* en un modelo es posible si éste dispone de constantes genéricas referentes al tiempo.

5.2.2.4 Mecanismo de configuración

La sintaxis general del mecanismo de configuración es:

```
configuration <nombre_configuracion> of <nombre_entidad> is
  for <nombre_arquitectura>
    { <configuración_de_componente> }
  end for;
end configuration;
```

Con <nombre_arquitectura> se especifica la arquitectura concreta que describe el funcionamiento del componente (en este caso del modelo principal). Por otro lado, con <configuración_de_componente> se puede configurar de manera individual cada uno de los componentes de una arquitectura estructural. Esta configuración no sólo consiste en indicar la arquitectura empleada para cada componente, sino que también se puede establecer el valor particular de las constantes genéricas del componente para cada instancia.

La utilización de configuraciones constituye un eficiente mecanismo para asociar una determinada a cada entidad. Puede ser útil para implementar la técnica de mutantes, como se comenta en el apartado 5.4.2.

5.3 Inyección mediante órdenes del simulador

Dentro de este apartado se pueden observar, a su vez, dos variantes: manipulación de señales y de variables.

Cuando se manipulan señales, los fallos se inyectan alterando el valor de las señales del modelo VHDL. Se hace desconectando la señal de su *driver* o *drivers* y forzando el nuevo valor. Cuando la inyección finaliza, se vuelve a conectar la ruta normal para esa señal.

Por otra parte, la manipulación de variables permite la inyección de fallos en los modelos comportamentales de los componentes, debido a la posibilidad de alterar el valor de las variables definidas en el código en VHDL. La secuencia de pasos que se deben seguir para la inyección de un fallo transitorio se podría especificar de la siguiente manera:

1. Simular hasta el instante de la inyección.
2. Desconectar el(los) *driver(s)* de la señal y asignarle el 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 que finalice el tiempo de simulación.

[52] Al menos así lo permite el simulador comercial Modelsim [ModelTech 2001a, ModelTech 2001b].

En el caso de querer inyectar un fallo permanente, se omitirían los pasos 3 y 4:

1. Simular hasta el instante de la inyección.
2. Desconectar el(los) *driver(s)* de la señal y asignarle el valor inyectado.
3. Simular hasta que finalice el tiempo de simulación.

Para inyectar fallos intermitentes, en el paso 5 habría que volver al 1, para repetir el proceso en la siguiente activación del fallo:

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

Por otra parte, la manipulación de variables permite la inyección de fallos en los modelos comportamentales de los componentes, debido a la posibilidad de alterar el valor de las variables definidas. Los pasos que hay que seguir son:

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

Es importante resaltar que, cuando se manipulan variables, no es posible controlar la duración de la inyección. Por tanto, no se pueden inyectar fallos permanentes.

Utilizando las órdenes del simulador, tanto sobre señales como sobre variables, se pueden inyectar algunos modelos de fallo significativos:

- *Stuck-at* '0' y '1'.
- *Open-line*.
- Inversión lógica (*bit-flip* en elementos de memoria y *pulse* en elementos combinacionales).
- *Indetermination*.
- Alteración de los retardos de propagación (*delay*).

Además, se pueden inyectar tanto fallos transitorios como permanentes.

Las principales ventajas que se argumentan para el uso de esta técnica de inyección son, por una lado que no es necesario modificar el código VHDL. Por otro, que permite el acceso en lectura y escritura a todas las señales y variables del modelo. No obstante, tiene el inconveniente de que depende totalmente de los simuladores comerciales existentes. De hecho, los pasos indicados para cada tipo de inyección están expresados en pseudolenguaje, y en función del simulador utilizado, se traduce en un conjunto de órdenes que se pueden agrupar en una *macro*, facilitando el proceso de inyección.

5.4 Inyección mediante modificación del modelo en VHDL

Como ya se ha comentado, dentro de este apartado se pueden distinguir dos técnicas. La primera de ellas se basa en añadir unos componentes específicos para la inyección de fallos, llamados **perturbadores** (*saboteurs*). La segunda se basa en la alteración de componentes existentes en el modelo VHDL, a los que se denomina **mutantes** (*mutants*).

5.4.1 Perturbadores

Un perturbador es un componente **añadido** al modelo original, con la misión de alterar el valor o las características temporales de una o más señales cuando se inyecta un fallo. Durante la operación normal permanece inactivo, y se activa sólo para inyectar el fallo. Las señales en las que se puede inyectar son las que conectan componentes. Es decir, esta técnica se aplica sobre modelos estructurales. En [Jenn *et al.* 1993b] se distinguen dos tipos de perturbadores, **en serie** y **en paralelo**.

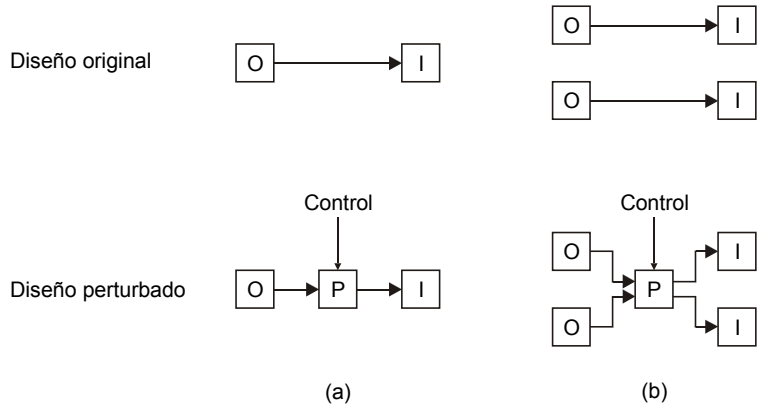

Figura 5.2: Estructura de un perturbador en serie [Jenn *et al.* 1993b]. (a) Simple. (b) Complejo.

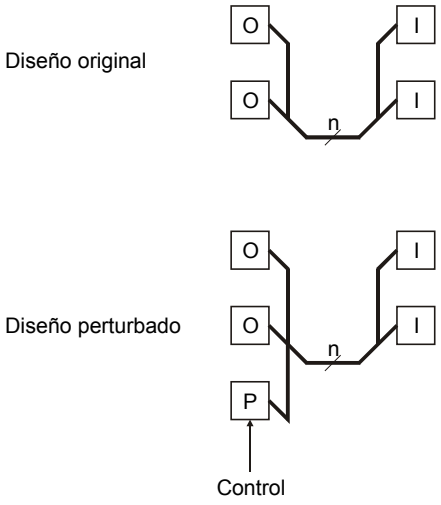

Figura 5.3: Estructura de un perturbador en paralelo [Jenn *et al.* 1993b].

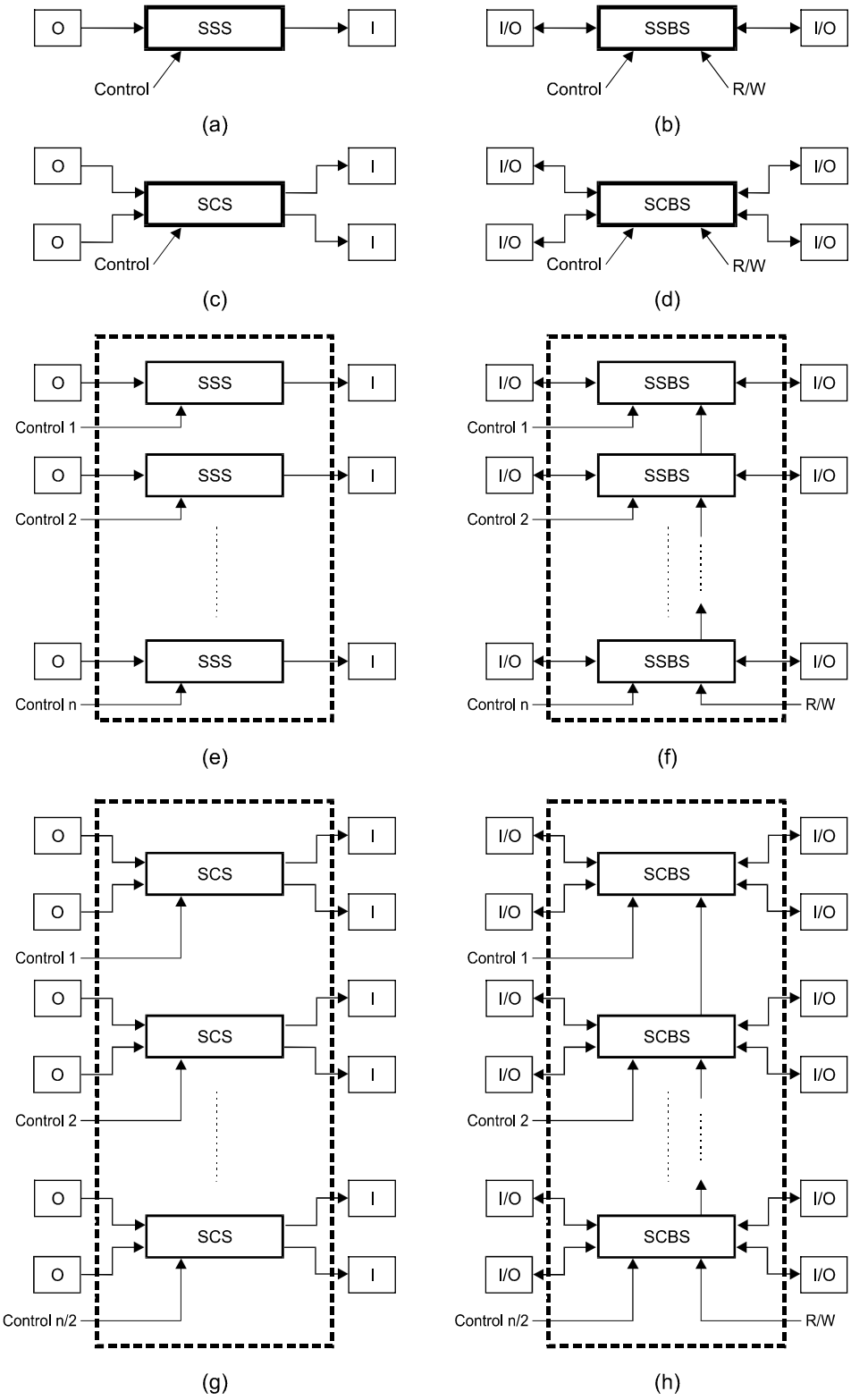

Figura 5.4: Perturbadores en serie propuestos en [Gracia *et al.* 2001b]. (a) Perturbador en Serie Simple; (b) Perturbador en Serie Simple Bidireccional; (c) Perturbador en Serie Complejo; (d) Perturbador en Serie Complejo Bidireccional; (e) Perturbador en Simple Unidireccional de *n* bits; (f) Perturbador Simple Bidireccional de *n* bits; (g) Perturbador Complejo Unidireccional de *n* bits; (h) Perturbador Complejo Bidireccional de *n* bits.

Los perturbadores en serie son componentes VHDL que se interponen entre una o varias salidas y sus correspondientes entradas. Cuando el número de entradas y salidas es 1, el perturbador en serie se denomina **simple**, y **complejo** cuando alguno de ellos es mayor que 1. En la Figura 5.2 se muestra el aspecto de estos perturbadores.

Los perturbadores en paralelo se implementan como un *driver* más en la generación de la señal resultante, de manera que, cuando se inyecta un fallo, su activación modifique el resultado de la función de resolución. En la Figura 5.3 se puede ver un ejemplo de perturbador en paralelo.

Esta es una clasificación muy simple. En la Figura 5.4 se muestra una extensión de los modelos de perturbadores en serie, presentada en [Gracia *et al.* 2001b]. Dichos modelos se aplican a señales bidireccionales y a buses, y se han desarrollado como parte del trabajo llevado a cabo para la realización de la herramienta de inyección de fallos incluida en la presente tesis:

a) **Perturbador en serie simple** (en inglés ***Serial Simple Saboteur***, SSS): Interrumpe la conexión entre una salida (*driver*) y su correspondiente receptor (entrada), modificando el valor recibido.

b) **Perturbador en serie simple bidireccional** (***Serial Simple Bi-directional Saboteur***, SSBS): Tiene dos puertos de entrada/salida, y una señal de lectura/escritura que determina la dirección de la perturbación (R/W).

c) **Perturbador en serie complejo** (***Serial Complex Saboteur***, SCS): Interrumpe la conexión entre dos salidas y sus correspondientes receptores, modificando los valores recibidos.

d) **Perturbador en serie complejo bidireccional** (***Serial Complex Bi-directional Saboteur***, SCBS): Tiene cuatro puertos de entrada/salida, y una señal de lectura/escritura que determina la dirección de la perturbación (R/W).

e) **Perturbador simple unidireccional de *n* bits** (***n-bit Unidirectional Simple Saboteur***, nUSS): Se usa para buses unidireccionales (direcciones y control) de *n* bits. Está compuesto por *n* perturbadores en serie simples (SSS).

f) **Perturbador simple bidireccional de *n* bits** (***n-bit Bi-directional Simple Saboteur***, nBSS): Se usa para buses bidireccionales (datos y control) de *n* bits. Está compuesto por *n* perturbadores en serie simples bidireccionales (SSBS).

g) **Perturbador complejo unidireccional de *n* bits** (***n-bit Unidirectional Complex Saboteur***, nUCS): Se usa para buses unidireccionales (direcciones y control) de *n* bits. Está compuesto por *n*/2 perturbadores en serie complejos (SCS).

h) **Perturbador complejo bidireccional de *n* bits** (***n-bit Bi-directional Complex Saboteur***, nBCS): Se usa para buses bidireccionales (datos y control) de *n* bits. Está compuesto por *n*/2 perturbadores en serie complejos bidireccionales (SCBS).

En cuanto a los perturbadores en paralelo, en [Gil 1999] se hace una aproximación a la implementación de este tipo de perturbadores, proponiendo dos métodos para realizar la inyección de forma efectiva:

- Modificar la función de resolución asociada a la señal a la que afectará el perturbador, para introducir el efecto prioritario del perturbador sobre el resultado de la resolución. Este método tiene el problema de que, a tenor de cómo se definen las funciones de resolución, es imposible que la función modificada pueda discernir de qué fuente procede

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, similar al `std_logic` (descrito en el apartado 5.2.2.1), 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 los debería utilizar el perturbador. De este modo, al especificar la función de resolución del nuevo tipo, al detectar la presencia de uno de los valores "prioritarios" el valor resuelto sería el valor "no prioritario" equivalente.

- Introducir un componente adicional que recibiera como entradas las diferentes fuentes de la señal afectada, además de la salida del perturbador. Esta segunda opción introduce un problema, ya que la solución propuesta es equivalente a desconectar realmente la señal de sus fuentes, por lo que la implementación es realmente la de un perturbador en serie.

Dada la elevada dificultad que la única solución aparentemente viable conlleva con vistas a su utilización en una herramienta de inyección, que recordemos que es el objetivo principal del presente trabajo de tesis, no se ha profundizado más en este tipo de perturbadores, y en adelante sólo se referirá a los perturbadores en serie.

En todos los modelos expuestos en la Figura 5.2 y en la Figura 5.4, la función de la entrada *Control* es la temporización de la inyección: su activación determina tanto el instante de inyección (t_{iny}) como la duración del fallo (Δt_{iny}). En la Figura 5.5 se puede comprobar con más claridad.

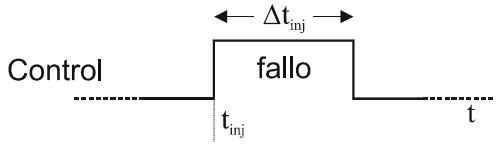

Figura 5.5: Temporización de la inyección de fallos.

Para especificar el modelo de fallo que se inyecta, se pueden seguir dos estrategias:

1. Incluir en el diseño del perturbador (al menos) una señal adicional que codifique el modelo de fallo inyectado.
2. Utilizar variables globales (véase el apartado 5.2.1). Esto simplifica tanto el modelo en VHDL del perturbador como las modificaciones que hay que introducir en el diseño. Sin embargo, tiene el inconveniente de que se pierde la estructuralidad de los diseños.

Utilizando perturbadores se pueden inyectar los siguientes tipos de fallo [Gil *et al.* 1998c]:

- *Stuck-at* '0' y '1'.
- *Open-line*.
- *Stuck-open*.
- Inversión lógica, tanto en circuitos de almacenamiento[53] (*bit-flip*) como combinacionales (*pulse*).

[53] Memorias y registros.

- *Indetermination*.
- *Delay*.
- *Short*.
- *Bridging*, según se define en [Jenn 1994a] (véase el capítulo 4).

Obsérvese que el número de modelos de fallos que se pueden utilizar en esta técnica es superior al de la técnica basada en las órdenes del simulador.

Por contra, el uso de perturbadores para inyectar fallos tiene el inconveniente de que, para poder especificar los diferentes tipos de fallo que se pueden aplicar en cada perturbador, es preciso añadir al modelo una serie de señales de control (véase de la Figura 5.2 a la Figura 5.4) para indicar la activación de un perturbador u otro, y el tipo de perturbación introducida en cada caso. Esto añade una complicación adicional al modelo y a la técnica.

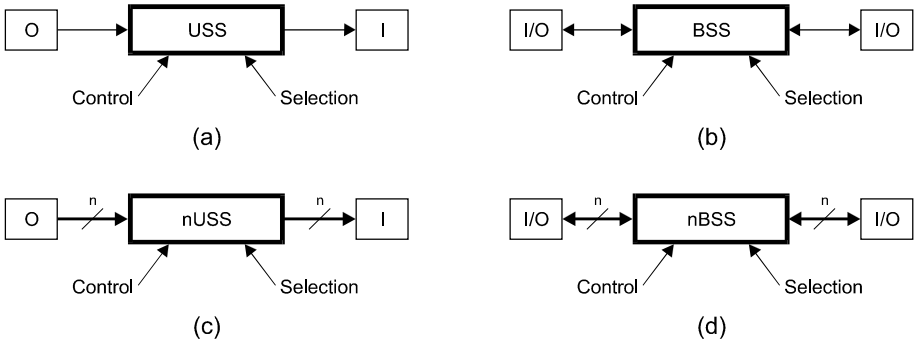

Figura 5.6: Nuevos perturbadores en serie propuestos. (a) Perturbador en Serie Unidireccional; (b) Perturbador en Serie Bidireccional; (c) Perturbador en Serie Unidireccional de *n* bits; (d) Perturbador en Serie Bidireccional de *n* bits.

Los ocho modelos de perturbadores de la Figura 5.4 se pueden simplificar en los cuatro mostrados en la Figura 5.6, ya que los seis modelos de perturbadores de más de un bit se pueden generalizar en dos de *n* bits: uno unidireccional y otro bidireccional:

a) **Perturbador en serie unidireccional** (en inglés ***Unidirectional Serial Saboteur***, USS): Es el perturbador en serie simple de la Figura 5.4–a, si bien permite inyectar más modelos de fallos que aquél.

b) **Perturbador en serie bidireccional** (***Bi-directional Serial Saboteur***, BSS): Es similar al perturbador en serie simple bidireccional de la Figura 5.4–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 en serie unidireccional de n bits** (***n-bit Unidirectional Serial Saboteur***, nUSS): Este modelo reemplaza a los tres modelos unidireccionales de la Figura 5.4.

d) **Perturbador en serie bidireccional de n bits** (***n-bit Bi-directional Serial Saboteur***, nBSS): Sustituye a los tres modelos bidireccionales de la Figura 5.4.

Además, en estos nuevos modelos se han introducido algunos cambios con respecto a los propuestos en [Gracia *et al.* 2001b]:

1. Se ha aumentado el número de modelos de fallos implementados. En particular, se ha incluido la distinción entre *bit-flip* (para elementos de memoria) y *pulse* (para circuitos combinacionales), considerada en el capítulo 4.
2. En los modelos bidireccionales se ha eliminado la entrada de control de sentido de la inyección, realizando este control el propio perturbador.
3. Se ha optimizado la implementación de las versiones comportamentales, simplificando la tarea de diseño.
4. La selección del fallo inyectado se realiza a través de una única señal de control, denominada *Selection*[54]. Como se podrá ver en el apartado 5.4.1.5, introduciendo un sutil cambio en los modelos de perturbadores, la modificación automática de un diseño mediante la inclusión de perturbadores se simplificará notablemente.

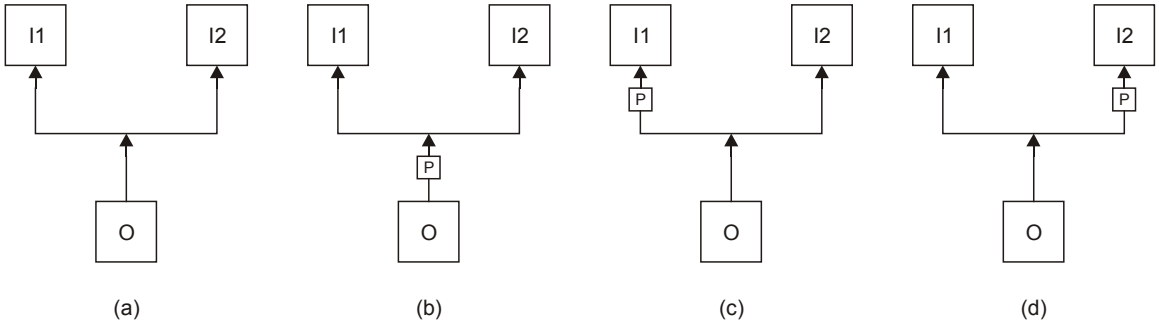

Figura 5.7: Estrategia de inserción de perturbadores [Boué *et al.* 1996].

Un aspecto importante de esta técnica de inyección lo constituye la estrategia de ubicación de los perturbadores. Dado el ejemplo simple de arquitectura estructural de la Figura 5.7–a, se pueden insertar perturbadores en tres lugares, y los resultados de inyectar fallos con ellos son completamente distintos:

- En la salida del módulo "O" (Figura 5.7–b): Este perturbador se correspondería con la existencia de fallos en la salida del componente "O", que afectarían por igual a todos los módulos que la recibieran como entrada.
- En la entrada del módulo "I1" (Figura 5.7–c): Mediante este perturbador se simula la existencia de fallos en la entrada del componente "I1", que sólo le afectarán a él.
- En la entrada del módulo "I2" (Figura 5.7–d): De manera análoga al caso anterior, aquí se simulan fallos en la entrada de "I2", que sólo afectan a este componente.

Para poder inyectar un mayor número de fallos, suele ser necesario insertar perturbadores en diferentes ubicaciones. En el caso del ejemplo, en las tres mostradas en la figura. Como

[54] En los modelos propuestos en [Gracia *et al.* 2001b]., la selección del modelo de fallo se realiza mediante tres líneas del tipo `std_logic`. En la nueva propuesta, sólo se emplea una, de tipo `integer`. De este modo se logra, por un lado, reducir el número de señales que controla. Por otra parte, se permite incluir un mayor rango de modelos de fallo.

serio inconveniente, se puede ver que el modelo modificado se complica mucho, si bien la automatización de la inserción es sencilla (véase el apartado 5.4.1.5).

A continuación se muestra la implementación de los modelos de perturbadores propuestos en la Figura 5.6 (apartados 5.4.1.1 a 0). Por último, en el apartado 5.4.1.5 se realiza un análisis de la problemática que surge a la hora de automatizar la inyección de fallos utilizando perturbadores. Se consideran dos aspectos principales: la sobrecarga introducida en el sistema original por la inserción de los perturbadores y la capacidad de modelado en cuanto a multiplicidad de fallos.

5.4.1.1 Perturbador en serie unidireccional

Como muestra la Figura 5.6–a, además de las señales *I* (entrada) y *O* (salida), este perturbador dispone de dos señales encargadas de realizar la inyección, que se pueden manejar desde el simulador mediante órdenes. Una de ellas (*Control*), tiene como misión iniciar la inyección, y su activación determina el instante de inyección (t_{iny}) y la duración del fallo (Δt_{iny}). La otra (*Selection*) se utiliza para codificar el modelo de fallo que se ha de inyectar. En el caso que nos ocupa, los modelos que se podrían aplicar son *stuck-at* '0', *stuck-at* '1', *bit-flip*, *pulse*, *open-line*, *delay*, *indetermination* y *stuck-open*.

A continuación se puede ver una posible implementación del perturbador. Se trata de una implementación comportamental en la que un proceso (`process`) se encarga de detectar tanto las variaciones en la entrada *I* como en la señal *Control*. Cuando varía la señal *Control*, puede ser para iniciar la inyección (pasando a '1') o para finalizarla (cambiando a '0'). En el primer caso, habrá que actualizar la salida dependiendo del modelo de fallo que se inyecta. En el segundo, también en función del modelo de fallo inyectado, deberá procederse a la restitución de la señal original[55]. No se ha considerado la realización de una implementación estructural porque, dadas las diferencias de operación necesarias para la inyección de cada modelo de fallo la hacen excesivamente compleja.

Las variaciones en la entrada sólo deben ser consideradas en dos casos: mientras no se está inyectando un fallo o cuando se inyecta un fallo de tipo *delay*, ya que éste sigue afectando a la generación del nuevo valor.

La implementación comportamental simplificada (en pseudocódigo) del perturbador es la siguiente:

```
entidad perturbador_serie_unidireccional
  generic (Delay : time := 0 ns);      -- Retardo necesario para
                                       -- inyectar fallos Delay
  port (I : in std_logic
        O : out std_logic
        Control : in std_logic
        Selection : in integer)
fin

arquitectura comportamental del perturbador_serie_unidireccional
```

[55] Este detalle es especialmente importante para distinguir los modelos *pulse* y *bit-flip* (véase el capítulo 4, en concreto el apartado 4.3.3): en el primero (aplicado a puntos combinacionales del modelo), al desaparecer el fallo debe restaurarse la salida en función del valor actual. Por el contrario, en el segundo (aplicado a elementos de memoria), la salida cambia cuando se produce una escritura o un ciclo de refresco.

```
  proceso
    esperar a que cambie Control o se actualice I
    si cambia Control
      si Control = '1'                  -- Comienzo de la inyección
        O <= f_start(I, Selection)      -- Inyectar el fallo
      si no                             -- Fin de la inyección
        O <= f_end(I, Selection)        -- Desactivar el fallo
      fin si
    si se actualiza I
      si Control = '1'                  -- Hay inyección
        O <= f_faulty(I, Selection)     -- Refrescar la inyección
      si no                             -- No hay inyección
        O <= I                          -- Transmitir a la salida
      fin si
    fin si
  fin proceso
fin
```

Las funciones *f_start*, *f_end* y *f_faulty* se encargan de la modificación de la salida al manifestarse el fallo, mientras se mantiene, y al desaparecer su efecto, respectivamente. Se pueden implementar de una manera sencilla mediante sentencias `case`, cuyo selector sería la entrada *Selection*. En la Tabla 5.1 se muestran las expresiones que habría que asignar a la salida en función del modelo de fallo para cada una de las funciones.

Modelo de fallo	f_start	f_end	f_faulty
stuck-at '0'	'0'	I	
stuck-at '1'	'1'	I	
bit-flip	`not`(I)		
pulse	`not`(I)	I	
open-line	'Z'	I	
delay			I `after` delay, delay > 0
indetermination	'X'	I	
stuck-open	'0' `after` $t_{retention}$, $t_{retention} > 0$		

NOTA.- Las casillas vacías indican que no hay que llevar a cabo ninguna acción.

Tabla 5.1: Expresiones para cada función de inyección de los perturbadores en serie de un bit.

Como se puede apreciar en la tabla, para la inyección de fallos de tipo *delay* o *stuck-open*, es preciso indicar un valor (con el que se afectará a las nuevas transiciones o para indicar el tiempo de retención, respectivamente). Se puede especificar de dos maneras:

1. Utilizando constantes genéricas (`generic`). Estas constantes tienen la particularidad de poder ser modificadas mediante órdenes en tiempo de simulación (al menos en Modelsim®), de forma similar a como se haría con una variable. Por ello, ésta ha sido la

solución adoptada en el ejemplo. Para simplificar el modelo, se ha incluido una única constante (Delay), que se aplicará en ambos casos.

2. Utilizando variables globales, como ya se indicó para la entrada *Selection*. Esta solución presenta los mismos inconvenientes que los vistos para aquélla.

5.4.1.2 Perturbador en serie bidireccional

Este perturbador (cuyo diagrama se muestra en la Figura 5.6–b) es muy similar al unidireccional, con la salvedad de que las dos líneas implicadas en la trasferencia de datos son bidireccionales. Los modelos de fallo que se pueden inyectar con este perturbador son los mismos que en el USS.

A continuación se muestra una implementación comportamental del perturbador en serie bidireccional. Un aspecto interesante de este modelo es su capacidad de decidir en qué sentido efectúa la inyección de los fallos. Su opción es siempre inyectar los fallos en el sentido en el que fluyen los datos en el instante de la inyección. Para ello, debe conocer en cada momento cuál ha sido el último puerto modificado.

```
entidad perturbador_serie_bidireccional
  generic (Delay : time := 0 ns);      -- Retardo necesario para
                                       -- inyectar fallos Delay
  port (IO1 : inout std_logic := 'Z';
        IO2 : inout std_logic := 'Z';
        Control : in std_logic;
        Selection : in integer);
end

arquitectura comportamental del perturbador_serie_bidireccional
  proceso
    esperar a que cambie Control o se actualicen IO1 o IO2
    si cambia Control
      si Control = '1'                 -- Comienzo de la inyección
        si el sentido de la transferencia es hacia IO1
          IO1 <= f_start(I, Selection) -- Inyectar el fallo en IO1
        si no    -- el sentido de la transferencia es hacia IO2
          IO2 <= f_start(I, Selection) -- Inyectar el fallo en IO2
        fin si
      si no                            -- Fin de la inyección
        IO1 <= f_end(IO2, Selection)   -- Desactivar el fallo
        IO2 <= f_end(IO1, Selection)
      fin si
    si se actualiza IO1
      si Control = '1'                 -- Hay inyección
        IO2 <= f_faulty(IO1, Selection)-- Refrescar la inyección
      si no                            -- No hay inyección
        IO2 <= IO1                     -- Transmitir a la salida
      fin si
    si se actualiza IO2
      si Control = '1'                 -- Hay inyección
        IO1 <= f_faulty(IO2, Selection)-- Refrescar la inyección
      si no                            -- No hay inyección
        IO1 <= IO2                     -- Transmitir a la salida
```

```
      fin si
    fin si
  fin proceso
fin
```

Los modelos de fallo que se pueden inyectar con este perturbador son los mismos que en el *perturbador en serie unidireccional*, por lo que las funciones *f_start*, *f_end* y *f_faulty* son las mismas que para éste. Las expresiones de asignación a la salida para cada función son, pues, las mostradas en la Tabla 5.1.

5.4.1.3 Perturbador en serie unidireccional de n bits

Este perturbador (en la Figura 5.6–c) es muy similar al unidireccional de un bit. De hecho, como se puede comprobar, el código es muy similar al de éste. A continuación se muestra una implementación comportamental del perturbador (en pseudocódigo), que únicamente permite inyectar fallos simples.

```
entidad perturbador_serie_unidireccional_de_n_bits
  generic (N : integer := 2;           -- Permite especificar un
                                       -- tamaño genérico del
                                       -- perturbador
           Delay : time := 0 ns);      -- Retardo necesario para
                                       -- inyectar fallos Delay
  port (I : in std_logic_vector(N-1 downto 0);
        O : out std_logic_vector(N-1 downto 0);
        Control : in std_logic;
        Selection : in integer);
fin

arquitectura comportamental del
             perturbador_serie_unidireccional_de_n_bits
  proceso
    esperar a que cambie Control o se actualice I
    si cambia Control
      si Control = '1'                 -- Comienzo de la inyección
        O <= f_start(I, Selection)     -- Inyectar el fallo
      si no                            -- Fin de la inyección
        O <= f_end(I, Selection)       -- Desactivar el fallo
      fin si
    si se actualiza I
      si Control = '1'                 -- Hay inyección
        O <= f_faulty(I, Selection)    -- Refrescar la inyección
      si no                            -- No hay inyección
        O <= I                         -- Transmitir a la salida
      fin si
    fin si
  fin proceso
fin
```

Como se puede apreciar, las diferencias básicas con el código correspondiente al *perturbador en serie unidireccional* son tres.

En primer lugar, en la declaración en la entidad de las líneas *I* y *O* se especifica que son de *n* bits. Se lleva a cabo mediante un genérico cuyo valor real para cada perturbador se especificará al integrarlo en el diseño perturbado. De este modo, un único modelo sirve para inyectar fallos sobre líneas de cualquier tamaño.

El conjunto de modelos de fallos aplicables es diferente al del perturbador de un bit, puesto que en el de *n* bits se incluyen los modelos *short* y *bridging*. En la Figura 5.8 se muestran los tipos de fallos *short* y *bridging* que se pueden inyectar con un perturbador en serie unidireccional de 4 bits. Obsérvese que, como ya se comentó en el capítulo 4, en el modelo de fallo *bridging* se ha seguido la definición expresada en [Jenn 1994a].

La diferencia fundamental en el código de los modelos de los perturbadores de un bit y de *n* bits estriba en la implementación de las funciones que manejan la inyección de los fallos: *f_start*, *f_end* y *f_faulty*. En la Tabla 5.2 se muestran las expresiones que habría que asignar a la salida en función del modelo de fallo para cada una de las funciones, para los perturbadores de *n* bits.

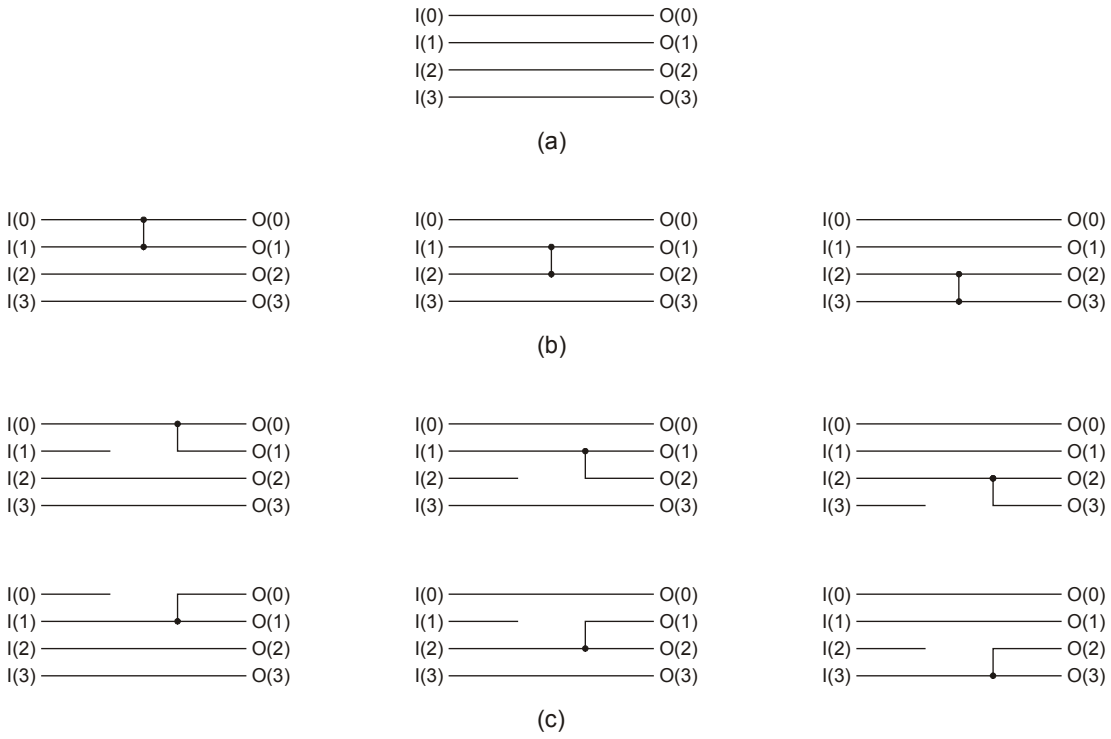

Figura 5.8: Tipos de fallos de los modelos *short* y *bridging* que se pueden inyectar con un perturbador en serie unidireccional de 4 bits. (a) Funcionamiento sin fallo. (b) Fallos *short*. (c) Fallos *bridging*.

Por último, los perturbadores de *n* bits deberían ser capaces de inyectar tanto fallos simples como múltiples (en los dominios del espacio y/o del tiempo). En el modelo propuesto sólo se ha considerado la inyección de fallos simples (como se indica en la Tabla 5.2). La problemática para considerar la inyección de fallos múltiples se discutirá en el apartado 5.4.1.5.

Modelo de fallo	f_start	f_end	f_faulty
stuck-at '0'	Para el bit con fallo, (i) O(i) <= '0' Para cada bit sin fallo, (j) O(j) <= I(j)	Para el bit con fallo, (i) O(i) <= I(i)	Para cada bit sin fallo, (j) O(j) <= I(j)
stuck-at '1'	Para el bit con fallo, (i) O(i) <= '1' Para cada bit sin fallo, (j) O(j) <= I(j)	Para el bit con fallo, (i) O(i) <= I(i)	Para cada bit sin fallo, (j) O(j) <= I(j)
bit-flip	Para el bit con fallo, (i) O(i) <= `not`(I(i)) Para cada bit sin fallo, (j) O(j) <= I(j)		Para cada bit sin fallo, (j) O(j) <= I(j)
pulse	Para el bit con fallo, (i) O(i) <= `not`(I(i)) Para cada bit sin fallo, (j) O(j) <= I(j)	Para el bit con fallo, (i) O(i) <= I(i)	Para cada bit sin fallo, (j) O(j) <= I(j)
open-line	Para el bit con fallo, (i) O(i) <= 'Z' Para cada bit sin fallo, (j) O(j) <= I(j)	Para el bit con fallo, (i) O(i) <= I(i)	Para cada bit sin fallo, (j) O(j) <= I(j)
delay			Para el bit con fallo, (i) O(i) <= I(i) `after` *Delay*, *Delay*>0 Para cada bit sin fallo, (j) O(j) <= I(j)
indetermination	Para el bit con fallo, (i) O(i) <= 'X' Para cada bit sin fallo, (j) O(j) <= I(j)	Para el bit con fallo, (i) O(i) <= I(i)	Para cada bit sin fallo, (j) O(j) <= I(j)
stuck-open	Para el bit con fallo, (i) O(i) <= '0' `after` $t_{retention}$, $t_{retention} > 0$ Para cada bit sin fallo, (j) O(j) <= I(j)		Para cada bit sin fallo, (j) O(j) <= I(j)
bridging	Para el bit con fallo, (i) O(i) <= I(i+1), o O(i) <= I(i-1) Para cada bit sin fallo, (j) O(j) <= I(j)		Para cada bit sin fallo, (j) O(j) <= I(j)
short	Para el bit con fallo, (i) O(i) <= I(i+1) y O(i+1) <= I(i), o O(i) <= I(i-1) y O(i-1) <= I(i) Para cada bit sin fallo, (j) O(j) <= I(j)		Para cada bit sin fallo, (j) O(j) <= I(j)

NOTA.- Las casillas vacías indican que no hay que llevar a cabo ninguna acción.

Tabla 5.2: Expresiones para cada función de inyección de los perturbadores en serie de *n* bits (para la inyección de fallos simples).

A la hora de inyectar un fallo simple, no sólo hay que especificar el tipo de fallo, sino también el bit sobre el que se aplica. Se han estudiado dos posibles soluciones:

- Indicar ambos datos en la selección del tipo de fallo (señal *Selection*) con un único valor entero. Esto implica que el gestor de la inyección de fallos será el encargado de especificar el bit afectado por la inyección.
- Añadir al perturbador la capacidad de generar aleatoriamente el bit sobre el que realizar la inyección del fallo.

En los modelos de perturbadores de *n* bits propuestos (el unidireccional y el bidireccional) se ha optado por la primera opción, ya que así se simplifica el diseño de los perturbadores, a costa de complicar la tarea de inyección.

5.4.1.4 Perturbador en serie bidireccional de n bits

Al igual que ocurre con el perturbador unidireccional de *n* bits, el bidireccional de *n* bits es una generalización del de un bit, considerando los detalles de implementación relativos al número de fallos inyectados y a cómo especificar los bits con fallo.

Del mismo modo, el modelo que aquí se presenta sólo considera la inyección de fallos simples. Una implementación comportamental (en pseudocódigo) del perturbador es:

```
entidad perturbador_serie_bidireccional_de_n_bits
  generic (N : integer := 2;           -- Permite especificar un
                                       -- tamaño genérico del
                                       -- perturbador
           Delay : time := 0 ns);      -- Retardo necesario para
                                       -- inyectar fallos Delay
  port (IO1 : in std_logic_vector(N-1 downto 0);
        IO2 : out std_logic_vector(N-1 downto 0);
        Control : in std_logic;
        Selection : in integer);
end

arquitectura comportamental del
             perturbador_serie_bidireccional_de_n_bits
  proceso
    esperar a que cambie Control o se actualicen IO1 o IO2
    si cambia Control
      si Control = '1'                 -- Comienzo de la inyección
        si el sentido de la transferencia es hacia IO1
          IO1 <= f_start(I, Selection) -- Inyectar el fallo en IO1
        si no    -- el sentido de la transferencia es hacia IO2
          IO2 <= f_start(I, Selection) -- Inyectar el fallo en IO2
        fin si
      si no                            -- Fin de la inyección
        IO1 <= f_end(IO2, Selection)   -- Desactivar el fallo
        IO2 <= f_end(IO1, Selection)
      fin si
    si se actualiza IO1
      si Control = '1'                 -- Hay inyección
        IO2 <= f_faulty(IO1, Selection)-- Refrescar la inyección
```

```
      si no                                -- No hay inyección
        IO2 <= IO1                         -- Transmitir a la salida
      fin si
    si se actualiza IO2
      si Control = '1'                     -- Hay inyección
        IO1 <= f_faulty(IO2, Selection)-- Refrescar la inyección
      si no                                -- No hay inyección
        IO1 <= IO2                         -- Transmitir a la salida
      fin si
    fin si
  fin proceso
fin
```

Como en el perturbador unidireccional de *n* bits, la Tabla 5.2 muestra las expresiones para las funciones que manejan la inyección propiamente dicha.

5.4.1.5 Automatización de la inserción de perturbadores

La inserción automática de los perturbadores en el modelo puede suponer, a priori, un esfuerzo considerable. Sin embargo, si se tiene en mente que esta tarea forma parte de una herramienta más completa, cabe pensar que la herramienta disponga de otras utilidades que pueden ayudar a la tarea de inserción de los perturbadores en el modelo. Una utilidad muy común en las herramientas de inyección de fallos mediante simulación de modelos en VHDL es un analizador lexicográfico (o *parser*), que permite extraer información del modelo a partir de sus ficheros fuente en VHDL. Esta información puede ser tan interesante como un árbol (lexicográfico) que contenga toda la estructura del modelo en forma de árbol, en el que se incluyen todas las sentencias y declaraciones.

Contando con esta información de partida, la inserción automática se simplifica notablemente, ya que el esfuerzo recaerá en una aplicación que sea capaz de recorrer el mencionado árbol lexicográfico, y hacer copias perturbadas de los ficheros fuente del modelo que incluyan los perturbadores. La inserción de los perturbadores implica tres acciones:

1. Añadir las señales necesarias para la activación de cada perturbador y la selección del modelo de fallo que se va a inyectar.
2. Insertar la declaración del componente tipo de cada perturbador que se introducirá en el diseño.
3. Añadir la instancia de cada perturbador, interponiéndose entre la señal del nivel correspondiente y el componente perturbado. Este paso implica la declaración de nuevas señales, para conectar el perturbador con el componente perturbado.

La Figura 5.9 muestra estos pasos con un ejemplo en el que se insertan algunos de los modelos de perturbadores propuestos. En ella se puede observar que, aunque hay que añadir declaraciones de señales en dos de los pasos, en la realidad sólo se hace en el primero de ellos, para poder escribir el código perturbado de forma correcta.

La tarea de inserción de los perturbadores está determinada por algunos detalles importantes, como son:

- Si la inserción se lleva a cabo de forma masiva o por demanda. En el primer caso, la aplicación encargada de insertar los perturbadores añadiría todos los perturbadores po-

sibles, en todos los niveles de la arquitectura estructural. Si la inserción se efectúa por demanda, es precisa la intervención del usuario para indicar a la aplicación los componentes que se desean perturbar.

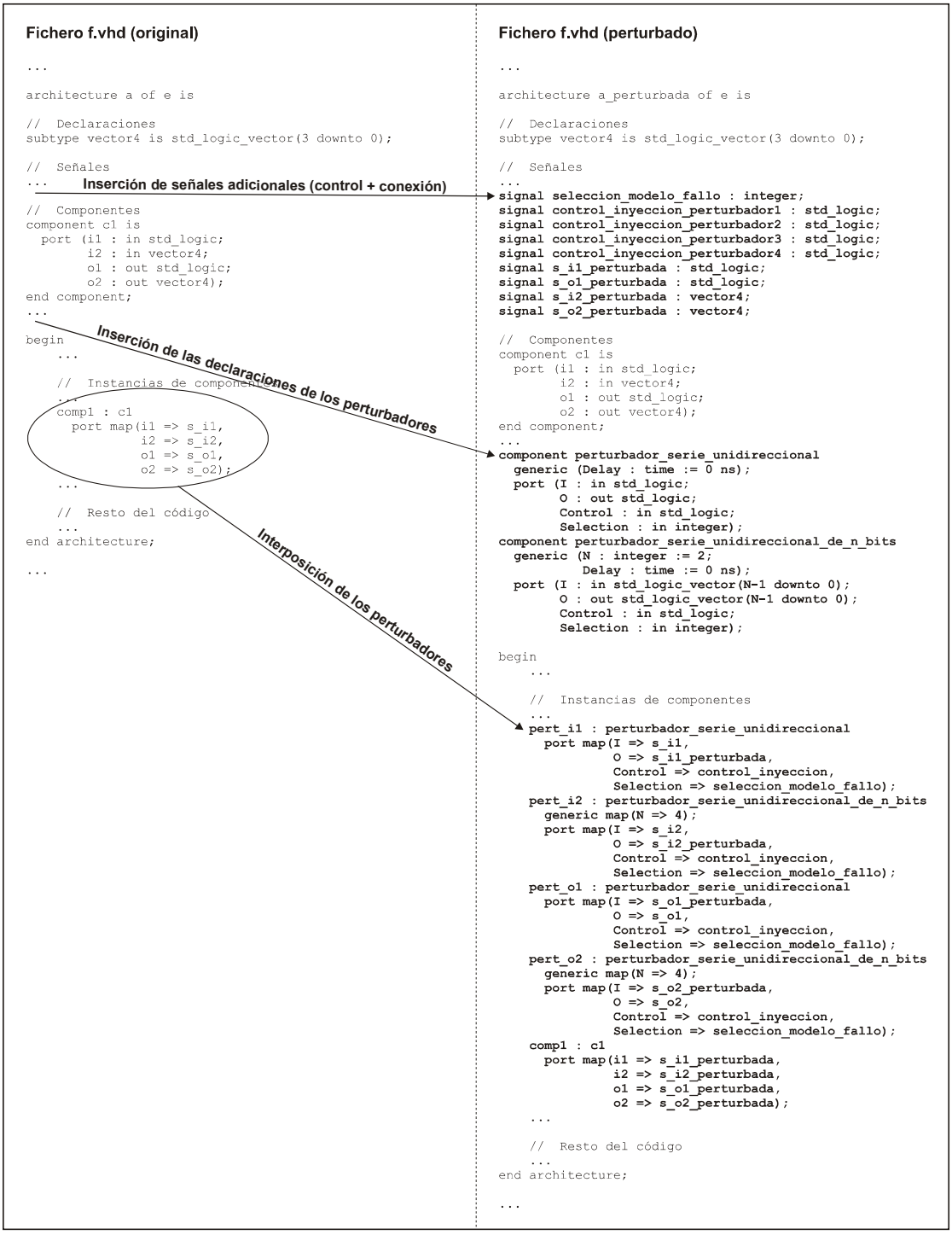

Figura 5.9: Automatización de la inserción de perturbadores en un modelo.

Evidentemente, la inserción masiva es mucho más sencilla de implementar, y además requiere una única compilación del modelo perturbado. Sin embargo, tiene el inconveniente de que, en modelos complejos (con muchos componentes), el modelo perturbado resultante sería de una complejidad mucho mayor, con el consiguiente deterioro de las prestaciones en cuanto al tiempo de simulación, debido no sólo al número de componentes adicionales, sino también de las señales añadidas (tanto para tareas de control como de conexión).

Por el contrario, mediante la inserción por demanda se puede reducir en gran medida la complejidad del modelo perturbado. Este método también tiene su parte negativa: cada vez que se configure una lista de perturbadores hay que generar el modelo perturbado y compilarlo, lo cual también implica un incremento en el coste temporal de la tarea de inyección.

- La clase de fallos que se van a inyectar:
  - ⇨ Fallos simples.
  - ⇨ Fallos múltiples:
    - En el tiempo, es decir, en el mismo lugar en diferentes instantes de tiempo:
    - En el espacio, o sea, en lugares diferentes en el mismo instante de tiempo. A su vez, los diferentes lugares pueden ser afectados por un mismo perturbador o por perturbadores distintos.
    - En el espacio y en el tiempo, que es una combinación de los dos anteriores.

  En los casos en los que se da multiplicidad en el tiempo hay que tener en cuenta si se permite el solapamiento de los fallos:

  - ⇨ Sin solapamiento. No puede inyectarse más de un fallo en un instante de tiempo.
  - ⇨ Con solapamiento. Se permite que en un instante de tiempo se inyecten fallos en diferentes lugares.

  La Figura 5.10 muestra los diferentes casos que se pueden dar con unos sencillos cronogramas donde se representa la activación en cada caso de las señales de control y de selección. Para simplificar los cronogramas, los nombres de las señales se han sustituido por otros más cortos que los mostrados en la Figura 5.9.

Como se puede comprobar en el ejemplo de la Figura 5.9, la inserción de perturbadores en un diseño ocasiona un fuerte impacto en el mismo. El motivo no es únicamente el número de componentes adicionales (los perturbadores) que se añaden. Además, por cada perturbador hay que incluir una señal para el control de la inyección y otra para conectarlo entre la línea original y el componente perturbado.

Por otro lado, como se deducirá en el apartado 5.4.1.6, para poder llevar a cabo la inyección de algunos de los tipos de fallos expuestos en la Figura 5.10, será preciso implementar modelos más complejos de perturbadores, así como añadir un número muy superior de señales al modelo original. De esta manera, cuando el número de perturbadores insertados sea muy elevado, la simulación del modelo perturbado (para la inyección de los fallos) se puede llegar a ralentizar en exceso.

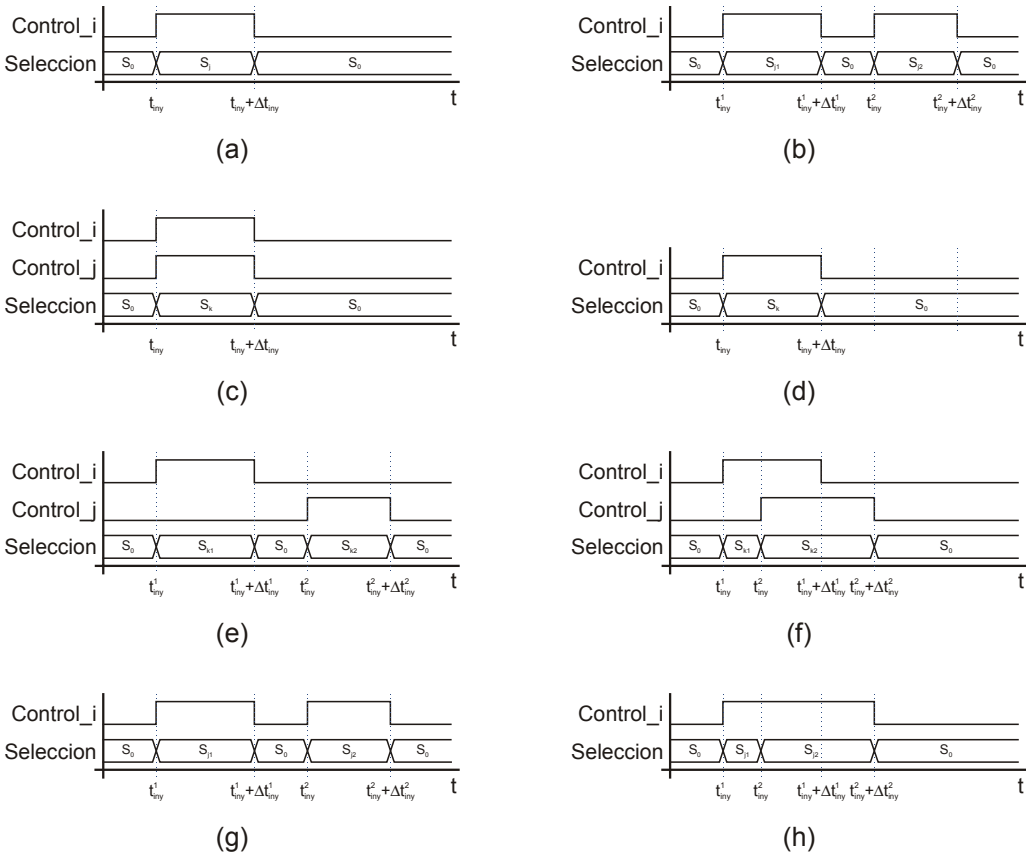

Figura 5.10: Multiplicidad de los fallos con perturbadores. (a) Fallos simples. (b) Fallos múltiples en el tiempo. (c) Fallos múltiples en el espacio (perturbadores diferentes). (d) Fallos múltiples en el espacio (mismo perturbador). (e) Fallos múltiples en el espacio y en el tiempo (en perturbadores independientes y sin solapamiento). (f) Fallos múltiples en el espacio y en el tiempo (en perturbadores independientes y con solapamiento). (g) Fallos múltiples en el espacio y en el tiempo (en el mismo perturbador y sin solapamiento). (h) Fallos múltiples en el espacio y en el tiempo (en el mismo perturbador y con solapamiento).

5.4.1.6 Análisis de la inserción de los modelos de perturbadores propuestos

A continuación se va a analizar la viabilidad de la implementación de los diferentes tipos de inyección de fallos expuestos en la Figura 5.10 utilizando los modelos de perturbadores propuestos. De este análisis se podrá ver que algunos de los casos no son directamente implementables, pero en la mayoría de los casos, lo serán modificando la implementación de los perturbadores.

El apartado finalizará con una recapitulación de las capacidades de las diferentes versiones de perturbadores propuestas, así como del coste que implica la implementación de cada una de ellas.

5.4.1.6.1 Inyección de fallos simples

Este es el más sencillo de los casos, y su implementación no plantea ningún problema. Sin embargo, observando el cronograma Figura 5.10-a, se puede ver que cada vez que se inyecte un fallo, la señal de control del perturbador implicado (*Control_i*) y la de selección del modelo de fallo (*Seleccion*) se activan y desactivan a la vez.

Como se trata de la inyección de fallos simples, esto sólo ocurrirá una vez en cada simulación del modelo. Por este motivo, en principio no es necesario distinguir el perturbador que se activa con una señal específica. En lugar de eso, se podría asignar esa misión a la propia señal de selección del modelo de fallo. En efecto, si el valor que se le asigna en cada inyección no sólo representa el modelo de fallo, sino también el perturbador implicado, la señal *Control* es innecesaria, y se puede suprimir.

Esta sencilla modificación, que se puede aplicar a los cuatro modelos de perturbadores propuestos (que en adelante se denominarán versiones normales), afectaría muy positivamente en la realización del diseño perturbado, ya que reduciría en un 33% el número de señales añadidas por cada perturbador (pasaría de tres a dos).

El inconveniente de esta nueva versión (que se denominará optimizada) es que es utilizable exclusivamente para la inyección de fallos simples, como se demostrará en próximos apartados.

5.4.1.6.2 Inyección de fallos múltiples en el tiempo

Siguiendo estrictamente la definición de este tipo de multiplicidad, ésta se puede dar exclusivamente en un mismo lugar en instantes de tiempo diferentes. En este sentido, en el cronograma de la Figura 5.10-b se muestra el caso más simple, en el que no hay solapamiento entre los fallos (que pueden ser iguales, como ocurriría en fallos intermitentes, o distintos). Sin embargo, es posible (aunque muy poco probable) que durante la ocurrencia de un fallo se pueda producir otro fallo diferente solapándose con el anterior (por ejemplo, son factibles combinaciones entre *delay* y *bit-flip*, o *delay* y *pulse*).

En cuanto a la posibilidad de implementación de este tipo de multiplicidad por los modelos propuestos, en caso de que no exista solapamiento entre los fallos, tanto las versiones normales como las optimizadas (obtenidas en el apartado anterior) son capaces de llevarla a cabo tal como se han propuesto. Por el contrario, ninguna de las dos versiones permite inyectar fallos solapados, como se puede ver en los cronogramas de la Figura 5.11:

- Las versiones normales (Figura 5.11-a), porque no es posible poner a '1' la señal *Control*, puesto que ya lo está. Además, este caso presenta una dificultad adicional: cuando *Control* pasa a '0' desactiva los dos fallos; entonces, ¿cómo se puede desactivar sólo uno de ellos?
- Las versiones optimizadas (Figura 5.11-b), porque el cambio de valor en la entrada *Selection* será tomado por el perturbador como el final de la inyección del fallo actual, además del inicio del siguiente.

Sin embargo, dada la baja probabilidad de que se produzca esta situación, no se van a buscar alternativas de implementación.

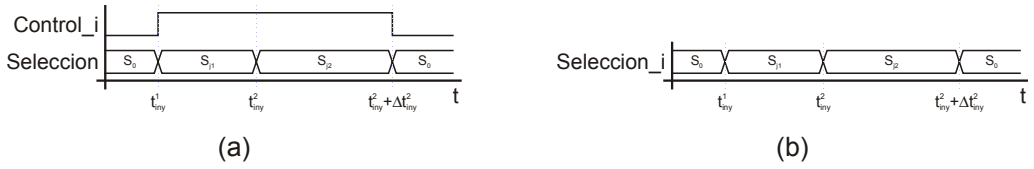

Figura 5.11: Problemas en la inyección de fallos múltiples en el tiempo solapados con los modelos de perturbadores propuestos. (a) Modelos normales. (b) Modelos optimizados.

5.4.1.6.3 Inyección de fallos múltiples en el espacio

Como se puede observar en los cronogramas Figura 5.10-c y Figura 5.10-d, se pueden considerar dos casos distintos, en función de si los fallos son manejados por perturbadores diferentes (que se correspondería con el caso general) o por un mismo perturbador. Este segundo caso se puede dar, por ejemplo, en fallos transitorios múltiples ocasionados por radiación[56], que pueden afectar a líneas de *bus* contiguas, o a bits contiguos de un mismo registro o palabra de memoria. Por las especiales características de esta situación, se analizarán los dos casos por separado.

Los modelos propuestos permiten efectuar la inyección de fallos múltiples en lugares independientes. Sin embargo, hay que considerar algunas particularidades:

- Sólo se puede realizar con las versiones normales; con las optimizadas (véase el apartado 5.4.1.6.1) no es posible, puesto que es necesario activar más de un perturbador en un mismo instante de tiempo, y utilizando exclusivamente la señal *Selection* para este menester, no es posible.
- En cambio, las versiones normales permiten además el caso especial de que los fallos inyectados tengan distinta duración, ya que cada perturbador se activa y desactiva con su propia señal *Control*.
- Existe una restricción general en la inyección de fallos múltiples en el espacio: puesto que sólo se utiliza una señal para especificar a todos los perturbadores el modelo de fallo que se debe inyectar, al activar simultáneamente más de un perturbador, **todos deberán inyectar el mismo fallo**. Para poder inyectar modelos de fallos diferentes, sería necesario declarar en el nivel principal tantas señales de selección como perturbadores, como sucede con la señal de control. En este caso, no sería preciso modificar los modelos de los perturbadores, sino el modelo perturbado. La sobrecarga introducida sería de aproximadamente un 33 % de señales adicionales. Puesto que se hará mención a esta posibilidad más adelante, a esta posible versión se la denominará con múltiples fallos.

En cuanto a la inyección de fallos múltiples a cargo de un mismo perturbador, en principio no es factible, pero es posible llevarla a cabo modificando la implementación de los perturbadores (evidentemente, sólo los de *n* bits). Esta modificación puede realizarse de tres maneras diferentes:

a) Codificando en el valor de *Selection* el modelo de fallo, el número de fallos inyectados y el bit (o bits) implicado(s). Como se recordará, en el modelo propuesto para fallos simples, ya se codifican el modelo de fallo y los bits implicados. Bastaría con asociar más valores numéricos para inyectar fallos a cada perturbador. Teniendo en cuenta que los enteros en VHDL se representan con 32 bits, el rango de valores diferentes que se pueden codificar con un número entero es muy elevado. A la implementación de los perturbadores utilizando esta solución se la llamará versión en múltiples bits.

b) Dotar al perturbador de la capacidad de decidir tanto el número de fallos que debe inyectar como los bits que deben ser afectados. Utilizando constantes genéricas, se podría configurar una serie de funciones de distribución (y sus parámetros correspondientes) que se utilizarían para esos menesteres. Evidentemente, en el momento de inyectar un

[56] Principalmente de iones pesados (véase el capítulo 4).

fallo, el perturbador haría uso de dichas constantes genéricas para generar aleatoriamente los objetivos buscados: el número de fallos y los lugares donde inyectarlos. En este caso, el valor de la entrada *Selection* sólo representa el modelo de fallo que se debe inyectar.

c) Dotar al perturbador de capacidad de decisión, pero sólo en cuanto a los bits afectados, especificando en *Selection* tanto el modelo de fallo como el número de fallos que se han de inyectar.

Las tres modificaciones propuestas son aparentemente igual de válidas. En opinión del autor, la primera es más coherente con la filosofía seguida en cuanto a la forma en que se lleva a cabo la inyección de fallos con perturbadores. Se parte de la base de que una herramienta externa de inyección de fallos decida aleatoriamente el instante de inyección, el número y modelo de los fallos que se van a inyectar, el instante en que se produce el fallo, y su duración. Entonces, la primera opción mantiene esta idea; en cambio, las otras dos, alterarían la forma de operar (el algoritmo de inyección), puesto que la herramienta perdería el control respecto a los lugares donde se inyectan los fallos. Este hecho, que puede llegar a ser contraproducente en cuanto a la realización de una herramienta de inyección, hace que sean descartadas a pesar de presentar unas posibilidades muy interesantes.

Al igual que ocurre con la inyección múltiple en lugares independientes, la solución adoptada no permite la inyección de fallos diferentes ni de distinta duración. Siguiendo con un ejemplo comentado anteriormente, con la solución actual (la denominada en múltiples bits) se simularía la ocurrencia de fallos múltiples ocasionados por una misma partícula radiactiva, (que en principio deberían producirse en localizaciones adyacentes). Sin embargo, las ocasionadas por diferentes partículas (o incluso por distintos mecanismos de fallo) en posiciones próximas, que no obligatoriamente adyacentes, no se podrían simular.

La solución a este problema es factible, aunque supone un elevado coste para el modelo perturbado. Siguiendo con la filosofía que ha conducido a la deducción de las versiones con múltiples fallos, los modelos de perturbadores de *n* bits se deberían modificar haciendo que las señales *Control* y *Selection* no fueran individuales, sino vectores del mismo tamaño que el perturbador. Es interesante advertir que esa nueva solución (a la que se denominará con múltiples fallos por perturbador) también resuelve el problema de la inyección múltiple dentro del mismo perturbador, tal como hace la versión en múltiples bits.

A la versión resultante de la aplicación conjunta de las versiones con múltiples fallos y con múltiples fallos por perturbador se la denominará versión completa.

5.4.1.6.4 Inyección de fallos múltiples en el espacio y en el tiempo

Los casos en que no se produce solapamiento en el tiempo (mostrados en los cronogramas Figura 5.10-e y Figura 5.10-g) no presentan ninguna dificultad para las versiones normales de los perturbadores. En cambio, cuando hay solapamiento de los fallos en el tiempo sí que hay que hacer algunas consideraciones:

- En la Figura 5.10-f se muestra el caso en que se inyecta un fallo mientras se está inyectando otro. El problema surge cuando se quiere desactivar el primero de los fallos inyectados (en $t = t^1_{iny} + \Delta t^1_{iny}$). Cuando esto suceda, el perturbador desactivará el fallo determinado por la entrada *Selection* en ese instante (S_{k2}), que no tiene por qué coincidir con el que se estaba inyectando desde (S_{k1}).

La solución a este problema es muy simple: basta con dotar al perturbador (a todos los modelos de las versiones normales) de "memoria". Esto es, en el momento en que se activa un perturbador, debe almacenar el valor de *Selection*, y cada vez que sea necesario, leer dicho valor almacenado, en lugar de la propia entrada.

En cambio, si se utilizan las versiones con múltiples fallos, este problema no se da, ya que como cada perturbador dispone de su propio par de entradas *Control* y *Selection*, no existe ambigüedad en el momento de la desactivación de los fallos.

- El cronograma de la Figura 5.10-h representa una situación similar a la anterior, pero en este caso dentro del mismo perturbador.

  La solución en este caso no es posible, salvo que se utilicen las versiones con múltiples fallos por perturbador de los perturbadores de *n* bits, ya que independizan la inyección sobre cada bit.

5.4.1.6.5 Recapitulación

En este apartado se comparan las diferentes versiones propuestas en los apartados anteriores. Los criterios de comparación que se utilizarán son la cantidad de fallos que permiten inyectar y del coste global de su implementación.

Las versiones deducidas son de dos clases. Por un lado están las versiones generales, que se refieren a todos los modelos de perturbadores; son cuatro: normal, optimizada, con múltiples fallos y completa. La otra clase de versiones son las específicas para los modelos de perturbadores de *n* bits: con múltiples fallos por perturbador y en múltiples bits.

En la Tabla 5.3 se muestra la comparación de las cuatro técnicas generales, junto con la aplicación de las dos específicas para los perturbadores de *n* bits a la versión normal. Así, la versión denominada con múltiples fallos por perturbador en la tabla se corresponde con la aplicación de dicha variante a la versión normal. Lo mismo sucede con la versión llamada en múltiples bits.

			Múltiples			
		Simples	**Tiempo**	**Espacio**	**Espacio y tiempo**	**Coste**
Versión	**Optimizada**	**Sí**	**Sí**[1]	No	No	Bajo
	Normal	Sí	Sí[1]	Sí[2]	Sí[1, 2, 3]	Medio
	Con múltiples fallos	Sí	Sí[1]	Sí[3]	Sí[1, 3]	Elevado
	En múltiples bits	Sí	Sí[1]	Sí[2, 4]	Sí[1, 2, 4]	Medio
	Con múltiples fallos por perturbador	Sí	Sí[1]	Sí[4]	Sí[1, 4]	Elevado
	Completa	Sí	Sí[1]	**Sí**	**Sí**[1]	Muy elevado

[1] Sin solapamiento temporal en una misma línea.
[2] Sólo para un mismo modelo de fallo.
[3] Sólo entre perturbadores diferentes.
[4] Sólo dentro de un mismo perturbador.

Tabla 5.3: Comparación de las diferentes versiones de perturbadores propuestas.

Como se puede comprobar, hay grandes diferencias en cuanto a las prestaciones y el coste de las diferentes opciones. Desde la simple versión optimizada (ideal para inyectar fallos simples, o a lo sumo fallos múltiples en el tiempo sin solapamiento), hasta la sofisticada y compleja versión completa (que permite la inyección de todas las posibilidades expuestas en la Figura 5.10, exceptuando el solapamiento de inyecciones en el tiempo en una misma posición individual), hay un amplio abanico de posibilidades, eso sí restringiendo alguna de las capacidades de inyección.

Se tratará de alcanzar un compromiso (por parte de quien deba inyectar los fallos) entre las prestaciones y el coste necesario para su realización. Evidentemente, tanto si la inserción de los perturbadores en el modelo se hace bajo demanda o si se lleva a cabo de forma masiva, será tarea de la herramienta de inyección interactuar con el usuario para configurar las prestaciones requeridas, y en función de ellas poder insertar los perturbadores más adecuados para su realización.

En cualquier caso, la herramienta de inyección debería disponer de todas las versiones de los modelos de perturbadores. Asimismo, debería ser capaz de implementar distintos algoritmos para la inyección de los fallos.

5.4.1.7 Modelos de fallos en perturbadores

Un aspecto crucial que no se ha mencionado hasta ahora es el que hace referencia a los modelos de fallos desde un punto de vista general. Con la descripción de los distintos modelos de perturbadores expuestos, se ha incluido una relación de los modelos de fallos que se pueden implementar con ellos. Esta relación se ha limitado en todos los casos a contemplar la perturbación de líneas (o vectores de líneas) de tipo `std_logic`, que es el tipo de dato básico en diseños estructurales al nivel de puerta.

Sin embargo, se olvida un detalle muy importante: VHDL permite modelar los diseños desde cualquier nivel de abstracción, y tanto desde un punto de vista estructural como comportamental. Esto significa que, en diseños estructurales realizados con un nivel de abstracción no tan bajo, es posible encontrar que para conectar componentes se emplean señales más complejas que las simples líneas digitales: booleanos, numéricos, mensajes, etc., o vectores de ellos.

Por este motivo, habría que generalizar el diseño de los perturbadores para que fueran válidos para cualquier tipo (o al menos limitando a un conjunto de ellos que cubra completamente otros niveles de abstracción), y así simplificar la automatización de las tareas de inserción y manejo durante la inyección.

5.4.2 Mutantes

Son componentes que **reemplazan** a los originales. Un mutante se comporta exactamente igual que el elemento al que reemplaza, excepto cuando se requiere la inyección de un fallo, que es cuando el nuevo componente abandona su comportamiento normal (el del elemento sustituido) y presenta un comportamiento anómalo, para la inyección de un fallo. La mutación puede llevarse a cabo de varias formas:

1. Añadiendo perturbadores a descripciones estructurales o comportamentales de componentes.

2. Modificando descripciones estructurales mediante el reemplazo de componentes; por ejemplo, una puerta NAND puede reemplazarse por una puerta NOR.
3. Modificando manualmente descripciones comportamentales para conseguir modelos de fallos complejos y detallados.
4. Modificando automáticamente instrucciones en modelos comportamentales; por ejemplo, generando operadores erróneos o cambiando identificadores de variables. Esta aproximación es similar a las técnicas de mutación usadas en la validación del test de de *software*.

En cuanto a los modelos de fallos, en [Ghosh y Chakarborty 1991] y [Armstrong *et al.* 1992] se proponen un conjunto de modelos de fallos generados a partir de la modificación de las estructuras sintácticas del código VHDL:

1. Asignación errónea de variable y/o señal.
2. Valor constante para una función.
3. Ejecución de un bucle `for` en un intervalo de valores prohibido.
4. Forzar la condición de un bucle `while` a verdadero o falso.
5. Forzar la condición de una sentencia `if` a verdadero o falso.
6. Forzar el selector de una sentencia `case` a uno de los valores propuestos, o a otro diferente.
7. Alterar las cláusulas de retardo (`after`) de las asignaciones en las señales.
8. Modificar las instrucciones de sincronización (`wait`), modificando sus listas de activación, condición y `time-out`.
9. Modificar las listas de activación de los procesos.

Todas las posibles modificaciones mencionadas anteriormente se agrupan en ocho modelos de fallos [Armstrong *et al.* 1992]:

- *Stuck-then*
- *Stuck-else*
- *Assignment control*
- *Dead process*
- *Dead clause*
- *Microoperation*
- *Local stuck-data*
- *Global stuck-data*

En [Riesgo y Uceda 1996] se hace una clasificación de los modelos de fallos, orientándolos 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.

- Todos los fallos son permanentes (esta es una restricción muy fuerte). La razón es que este estudio está orientado al test.
- 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).

La clasificación propuesta distingue tres clases de fallos:

- Fallos en datos. Los fallos de esta clase dependen del tipo del dato. Se distinguen diferentes categorías de tipos de datos:
  - ⇨ Tipos digitales (`bit`, `std_logic`, etc.). Para estos sólo se consideran los modelos *stuck-at* ‘0’ y ‘1’.
  - ⇨ Tipos enumerados. En este caso, los fallos pueden ser fijar el valor (*stuck-at*) a cualquier valor del tipo.
  - ⇨ Tipos numéricos enteros (`integer`, `natural`, `positive`). Conociendo el número real de bits utilizado para su codificación, se puede plantear el modelo *stuck-at* ‘0’ y ‘1’ sobre cualquiera de los bits[57].
  - ⇨ Vectores n-dimensionales de los tipos anteriores. En este caso se considerará un único fallo en uno de los componentes del vector, en función del tipo base del mismo.
  - ⇨ No se permiten tipos `record`.
- Fallos en expresiones. En este caso se distingue entre expresiones de control (las utilizadas en las sentencias secuenciales de VHDL) y de datos (las que aparecen en sentencias de asignación). Los modelos de fallos en expresiones de datos son los mismos que en los fallos en datos. En cuanto a los fallos en expresiones de control hay que considerar los siguientes casos:
  - ⇨ Sentencia `if` (`if` condición `then` ... `else` ...). La condición se puede fijar a `true` (modelo *stuck-at true*) o a `false` (modelo *stuck-at false*).
  - ⇨ Sentencia `case` (`case` expresión `is when` ...). La expresión se puede fijar a cualquiera de sus posibles valores (modelo *stuck-at “all possible values”*).
  - ⇨ Bucle `for` (`for` índice `in` rango `loop` ...). Los fallos considerados en este caso consisten en modificar el rango de bucle, bien incrementando en 1 el valor superior o disminuyendo en 1 el valor inferior.
- Fallos en sentencias. En función de si las sentencias son concurrentes o secuenciales, se distinguen dos casos. Las sentencias concurrentes se deben convertir en su equivalente secuencial (mediante un proceso y las consiguientes sentencias secuenciales), y aplicarles los mismo modelos que a éstas.

[57] Obviamente, en este caso sería necesario hacer dos conversiones de valor: una del valor numérico al valor binario, y otra del valor binario mutado a su correspondiente valor numérico.

Los modelos de fallos de las diferentes sentencias secuenciales son:

- ⇨ Sentencia `if`. Se consideran dos fallos: no ejecutar las sentencias asociadas al `then` (modelo *dead-then*) o no ejecutar las asociadas al `else` (modelo *dead-else*).
- ⇨ Sentencia `case`. El fallo propuesto consiste en no ejecutar las sentencias correspondientes a la opción afectada (modelo *dead-alternative*).
- ⇨ Bucle `for`. El modelo propuesto en este caso es no ejecutar el cuerpo del bucle (modelo *dead-loop*).
- ⇨ Llamada a procedimiento (`procedure`). El fallo causa la no ejecución del procedimiento (modelo *dead-call*).
- ⇨ Asignaciones (de señal o variable). El fallo consiste en no realizar la asignación (modelo *dead-assignment*).

Aunque los modelos básicos (sobre todo en los fallos en datos de tipos digitales) son muy simples, y que no se consideran los fallos de tipo *delay*, esta clasificación puede servir como base para la realización de mutantes si en los fallos en datos de tipos digitales se añaden los modelos *indetermination*, *open-line*, *bit-flip*, *pulse* y *stuck-open*, y se permite alterar los retardos en las sentencias de asignación y de sincronización (`wait`).

Más recientemente, en [Leveugle y Hadjiat 2000], se propone un nuevo modelo de fallo para descripciones comportamentales de máquinas de estados (por ejemplo de la unidad de control de microprocesadores): es el modelo 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.

La realización de este modelo de fallo está cubierta por la descripción de [Riesgo y Uceda 1996], 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.

En un principio, la técnica basada en mutantes consiste en crear réplicas de las arquitecturas de los componentes del modelo, en cada una de las cuales se realiza una serie de modificaciones (mutaciones), y agregarlas al modelo original. Se puede aplicar tanto a diseños estructurales como a comportamentales. En un diseño estructural, la mutación consiste en asociar a un componente una arquitectura modificada. En un diseño comportamental, la mutación consiste en alterar alguna de las sentencias de los procesos de la arquitectura. En cualquier caso, resulta muy útil la utilización del mecanismo de configuración (sentencia *configuration* del lenguaje VHDL) para seleccionar la implementación utilizada en cada caso.

La inyección consistirá en, a través del mecanismo de configuración, elegir una combinación de las arquitecturas (mutadas o no) asociadas a los diferentes componentes del modelo, y simularla. Esta definición implica el modelado de fallos **permanentes** desde el comienzo de la simulación. Para la inyección de fallos transitorios o de fallos permanentes desde un instante de tiempo intermedio, habría que establecer un mecanismo de sincronización entre la simulación de la configuración sin fallos a la otra (y viceversa en el caso de fallos transitorios).

La mayor ventaja de esta técnica es que permite inyectar fallos a un alto nivel de abstracción. Sin embargo, tiene el inconveniente de que las modificaciones en el diseño original pueden conllevar un crecimiento desmesurado del código que puede hacerlo inviable en diseños

complejos. Esto sucede cuando se altera un número muy elevado de partes del diseño, puesto que pueden aparecer infinidad de mutantes diferentes de un mismo elemento (componente, *package*, etc.).

Existen algunas propuestas de implementación de esta técnica. En [Gil 1999] se describen dos posibles implementaciones:

- Para modelos comportamentales, o para modelos estructurales sin mutación de componentes.

  Según esta propuesta, mostrada en la Figura 5.12, hay que añadir al diseño original (compuesto por la entidad "nombre_entidad" y la arquitectura "sinfallos"), N arquitecturas asociadas a la misma entidad ("mutacion1 a "mutacionN") y N+1 configuraciones: una para asociar la arquitectura "sinfallos" a la entidad ("sinfallo"), y las otras N para asociar las N arquitecturas mutadas a la entidad ("mutante1" a "mutanteN").

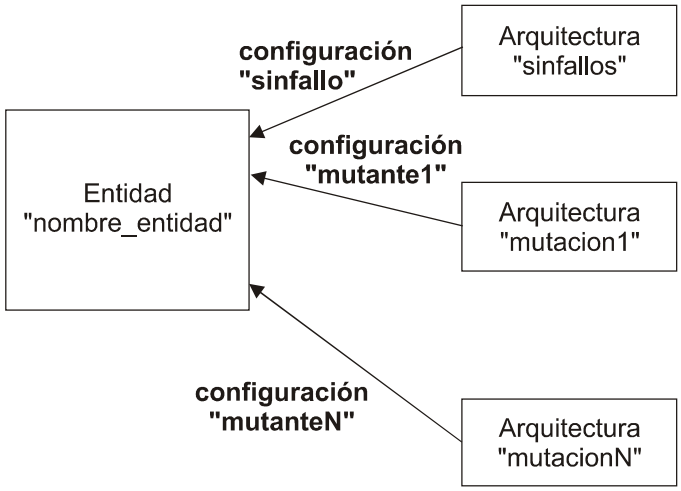

Figura 5.12: Implementación de mutantes mediante el mecanismo de configuración para modelos con arquitecturas comportamentales, o estructurales sin componentes.

En VHDL sería:

```
-- Declaración de la entidad
entity nombre_entidad
port (...)
end entity;

-- Declaración de las arquitecturas asociables a la entidad
architecture sinfallos of nombre_entidad is
  ...
begin
  ...
end architecture;

architecture mutacion1 of nombre_entidad is
  ...
begin
  ...
end architecture;
```

```
...

architecture mutacionN of nombre_entidad is
  ...
begin
  ...
end architecture;

-- Asociación de las diferentes arquitecturas a la entidad
-- mediante configuraciones
configuration sinfallo of nombre_entidad is
  for sinfallos
    ...
  end for;
end configuration;

configuration mutante1 of nombre_entidad is
  for mutacion1
    ...
  end for;
end configuration;

...

configuration mutanteN of nombre_entidad is
  for mutacionN
    ...
  end for;
end configuration;
```

- Para arquitecturas estructurales con mutación de componentes.

  Esta propuesta (mostrada en la Figura 5.13) parte de un diseño compuesto por la entidad "nombre_entidad" y la arquitectura "estructural", en la que se importan M componentes. La implementación de cada uno de estos componentes se realiza mediante una entidad y una arquitectura denominada de forma genérica "sinfallos".

  La aplicación de la técnica requerirá, en primer lugar, la generación de una serie de arquitecturas mutadas para cada uno de los M componentes (llamadas, para el componente i-ésimo, "mutante1" a "mutanteNi")[58]. En segundo lugar, se deben generar K+1 configuraciones, donde K es el número total de combinaciones posibles de utilización de las distintas arquitecturas mutadas.

  El valor de K depende de las restricciones que se pongan en cuanto al número de fallos que se quieren inyectar. Por ejemplo, si sólo se permiten fallos simples (es decir, en ca-

[58] Obsérvese la coincidencia de nombres de las diferentes arquitecturas ("sinfallos", "mutante1", etc.). En VHDL no existe ambigüedad por este hecho, ya que el nombre real de una arquitectura viene determinado, tanto por el nombre de la arquitectura como por el de la entidad, que sí es único. En el código VHDL, concretamente en las configuraciones, puede distinguirse perfectamente cómo cada par entidad-arquitectura se asocia de forma unívoca, sin ambigüedades.

da configuración sólo hay un componente cuya arquitectura sea mutada), el valor de K está definido por:

$$K = \sum_{i=1}^{M} Ni$$

La (K+1)-ésima configuración es aquélla en la que se utilizan las arquitecturas originales, esto es, sin fallos (denominada "sinfallo").

El código VHDL con el que se realizaría sería:

```
-- Declaración de la entidad
entity nombre_entidad
  port (...)
end entity;

-- Declaración de la arquitectura estructural
architecture estructural of nombre_entidad is
  -- Declaración de los componentes del nivel inferior
  component C1
    port (...)
  end component;
  component C2
    port (...)
  end component;
  ...
  component CM
    port (...)
  end component;
  -- Otras declaraciones
  ...
begin
  ...
  -- Instanciación de los componentes
  Componente1 : C1
    port map (...);
  Componente2 : C2
    port map (...);
  ...
  ComponenteM : CM
    port map (...);
  ...
end architecture;

-- Asociación de las diferentes arquitecturas a cada componente
configuration sinfallo of nombre_entidad is
  for estructural
    for Componente1 : C1
      use entity work.C1(sinfallos);
    for Componente2 : C2
      use entity work.C2(sinfallos);
    ...
    for ComponenteM : CM
```

```
      use entity work.CM(sinfallos);
  end for;
end configuration;

configuration mutante1 of nombre_entidad is
  for estructural
    for Componente1 : C1
      use entity work.C1(mutante1);
    for Componente2 : C2
      use entity work.C2(sinfallos);
    ...
    for ComponenteM : CM
      use entity work.CM(sinfallos);
  end for;
end configuration;

...

configuration mutanteK of nombre_entidad is
  for estructural
    for Componente1 : C1
      use entity work.C1(sinfallos);
    for Componente2 : C2
      use entity work.C2(sinfallos);
    ...
    for ComponenteM : CM
      use entity work.CM(mutanteNM);
  end for;
end configuration;
```

Como ya se explicó anteriormente, la implementación de los mutantes siguiendo este método implica que en cada simulación sólo participa una determinada configuración (con o sin mutaciones), por lo que en principio se inyectan fallos permanentes desde el inicio de la simulación. Evidentemente, esta es una limitación muy importante, ya que imposibilita tanto la inyección de fallos transitorios como la de fallos permanentes que se produzcan una vez comenzada la simulación.

Para evitar este problema, en [Gil 1999] se sugiere un método de implementación dinámica de mutantes. El método se basa en la utilización de lo que en VHDL se denominan "señales guardadas[59]"(del inglés *guarded signal*) y del mecanismo de configuración. Utilizando un conjunto de órdenes del simulador, es posible conmutar dinámicamente la simulación de varios bloques, y las arquitecturas asociadas a los mismos. Estas órdenes (agrupadas en *macros* que de manera simbólica se pueden denominar *Guardar_Estado* y *Recuperar_Estado*) permiten guardar el estado del modelo en un fichero, y restaurar desde un fichero el estado del modelo. Por *estado* se entiende el tiempo de simulación y el contenido de todas las variables y señales del modelo. El diagrama temporal de la Figura 5.14 muestra cómo se lleva a cabo la inyección de fallos transitorios utilizando este método.

[59] O protegidas.

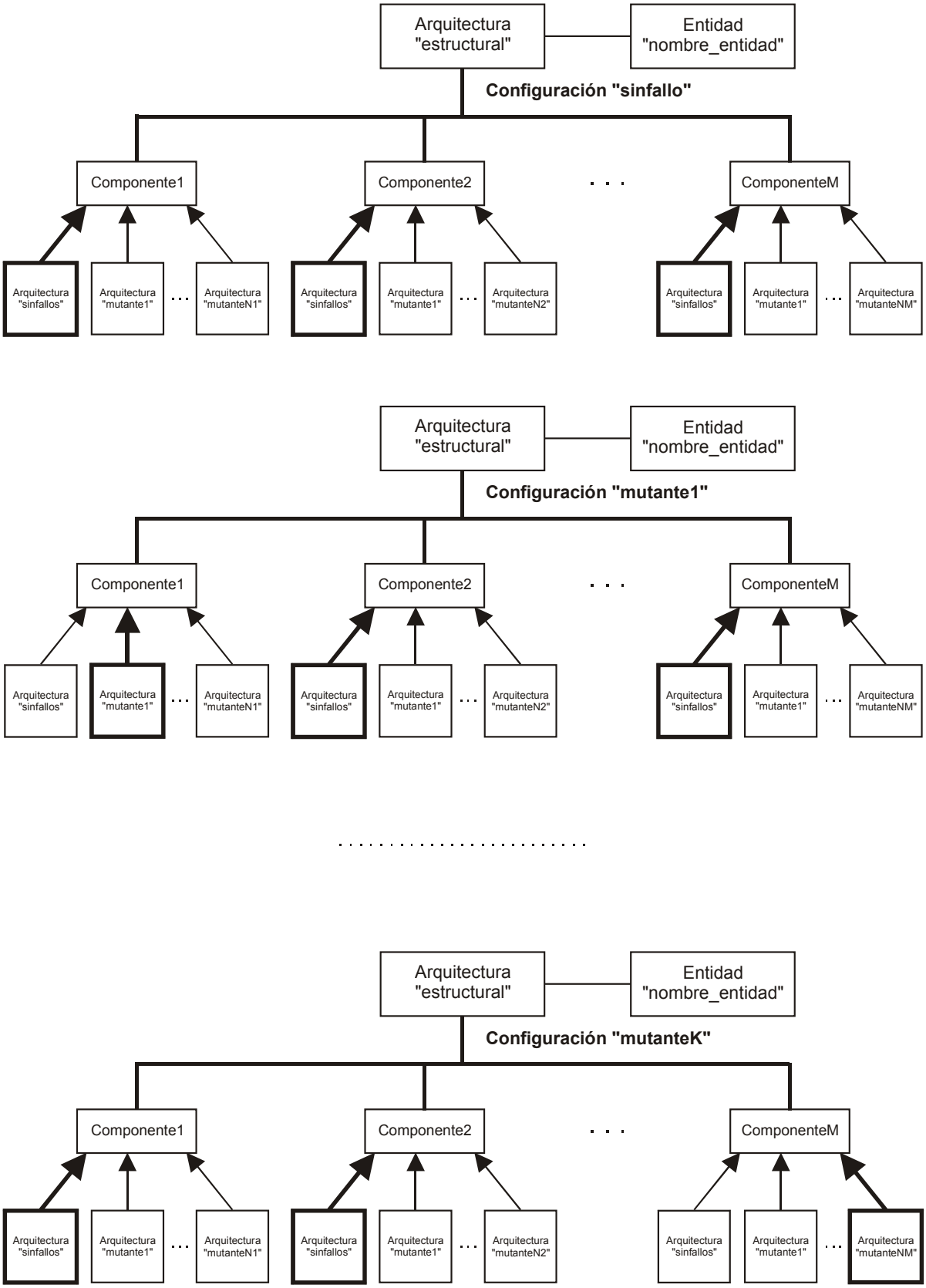

Figura 5.13: Implementación de mutantes mediante el mecanismo de configuración para modelos estructurales con mutación de componentes.

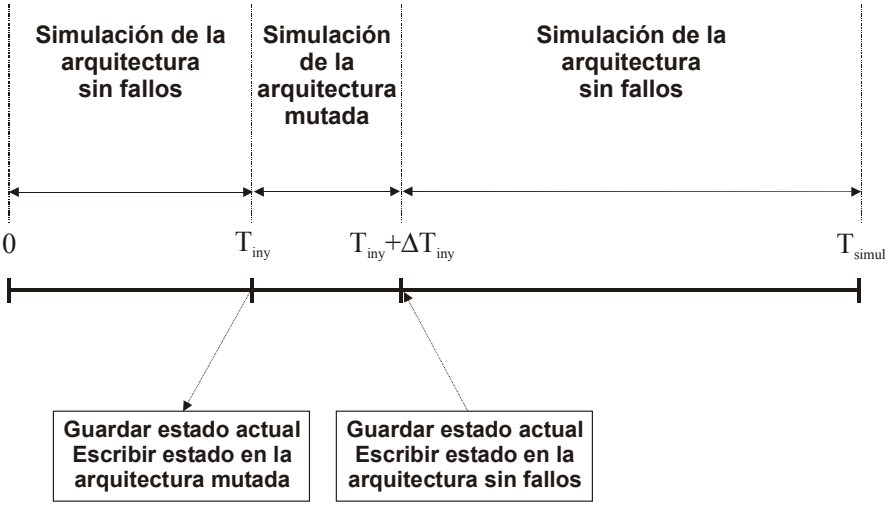

Figura 5.14: Diagrama temporal de la inyección de fallos transitorios con mutantes mediante señales guardadas.

Aunque esta técnica resulta muy atractiva desde un punto de vista teórico, presenta algunos problemas de implementación:

- Las operaciones de almacenamiento y carga del estado son muy costosas, y ralentizan en demasía las simulaciones correspondientes a las inyecciones de fallos, especialmente en el caso de los fallos transitorios (puesto que requieren dos, una para el inicio de la inyección y otra para la finalización). Además, los ficheros que contienen el estado del modelo ocupan mucho espacio.
- Hay restricciones de sincronización. Esto se debe a que para poder transferir el estado de una configuración a otra, ambas deben tener la misma estructura interna. Es decir, no puede haber discrepancias estructurales entre las diferentes arquitecturas: deben tener las mismas señales y variables (con igual nombre, tipo base y tamaño). Por esta razón, algunas de las mutaciones posibles no serán factibles.

Por todos estos motivos, esta técnica todavía no ha sido implementada en toda su capacidad, sino que se han hecho algunos intentos preliminares a nivel de prototipo, seleccionando un número reducido de partes de código modificables y efectuando un número limitado de inyecciones. Ejemplos de estos intentos se muestran en [Gracia *et al.* 2000, Gracia *et al.* 2001a, Gracia *et al.* 2001b].

Se puede pensar sin embargo en otro método de implementar los mutantes que, aunque no es demasiado elegante, permite inyectar cualquier tipo de fallo eludiendo los problemas de sincronización vistos anteriormente. Básicamente, la idea es reducir el coste temporal del proceso de inyección eliminando el proceso de conmutación. El precio que habría que pagar es la realización de un nuevo modelo que incluyera todas las posibles mutaciones.

El método se basa en parte en el mostrado en [Gil 1999], en el sentido de que utilizará el mecanismo de configuración para seleccionar las arquitecturas utilizadas para cada componente. La diferencia estriba en que, a diferencia de aquél, aquí se propone realizar una única mutación de cada arquitectura que contenga todas las posibles modificaciones, que se activarán en función de sentencias de selección (`if` y `case`) añadidas al modelo. En el modelo mutado, se deberá utilizar para todos los componentes la versión mutada de la arquitectura. La activación de uno u otro fallo se realizará en función de unas señales de control que se añadi-

rán a cada componente, y que se declararán en el diseño principal. Estas señales de control son similares a las utilizadas para los perturbadores, y su misión es seleccionar una mutación de entre todas las posibles.

En [Velazco *et al.* 2001, Civera *et al.* 2001a] se utiliza un método muy similar (explicado en el apartado 3.2.6). La técnica es muy similar en ambos casos, y se aplica a modelos estructurales a nivel de puerta. La mutación consiste en reemplazar los biestables originales por otros capaces de inyectar fallos. Como estos trabajos se corresponden con la técnica de inyección de fallos denominada *emulación de fallos con FPGA*, la inclusión "masiva" de mutantes en el modelo está limitada por el tamaño 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 del mismo modo que se expone en [Velazco *et al.* 2001, Civera *et al.* 2001a].

Los modelos de fallos que se proponen para este método son el compendio de todos los descritos al principio del apartado [Ghosh y Chakarborty 1991, Armstrong *et al.* 1992, Riesgo y Uceda 1996, Leveugle y Hadjiat 2000]. De este modo, el usuario tendrá un amplio abanico de posibilidades a la hora de seleccionar los tipos de mutaciones que desea para sentencia susceptible de ser modificada.

La gran pega de este método es que cada arquitectura mutante será muy compleja (y el tamaño de los ficheros puede dispararse), con la consiguiente complejidad de la simulación. Sin embargo, presenta cuatro ventajas muy interesantes:

1. Facilita la sincronización de la inyección, que se realizará exactamente igual que en los perturbadores, a través de órdenes del simulador para activar y desactivar las señales de control que determinan la inyección de un fallo.
2. La automatización del proceso de generación de las arquitecturas mutadas es relativamente sencilla, partiendo de la base de que cualquier herramienta de inyección es capaz de generar un árbol lexicográfico del modelo donde se contengan todas las sentencias (en particular aquéllas que son susceptibles de ser mutadas). A partir de este árbol, rescribir de manera automática un fichero fuente en VHDL con las modificaciones adecuadas es relativamente fácil.
3. La complejidad de modificación del modelo principal es tan simple como en los perturbadores: bastará con añadir las nuevas señales de control de los mutantes (de la inyección), modificar las declaraciones de los componentes (para utilizar los nuevos con las señales de control incorporadas), e integrar las arquitecturas mutadas, como se describe en el apartado 5.4.2.1.
4. El modelo mutado sólo necesita compilarse una vez, ya que incorpora todas las posibles mutaciones. Al igual que se comentó para los perturbadores, también podría plantearse la posibilidad de incorporar los mutantes por demanda, en función de las clases de sentencias a las que se desea afectar y de los modelos de fallos que se quiere utilizar. Esta selección podría ser tanto general como particular para cada sentencia susceptible de ser mutada. Al igual que sucedía con los perturbadores, la incorporación de mutantes por demanda exigirá la generación y compilación del modelo mutado mientras se

configura cada experimento de inyección, con el consiguiente deterioro de las prestaciones de la técnica. Será, pues, necesario alcanzar una solución de compromiso.

5.4.2.1 Generación automática de mutantes con el método propuesto

En este apartado se describe cómo se realizaría la mutación de un modelo siguiendo la propuesta realizada. Del mismo modo que en los perturbadores, el proceso de generación automática de la mutación del modelo se puede llevar a cabo, bien de forma masiva, o por demanda. La generación masiva de mutaciones implicará un crecimiento desmesurado del código, a tenor del gran número de modificaciones diferentes que se pueden introducir a una misma sentencia. Por el contrario, con una especificación selectiva de los modelos de mutantes a cada tipo de sentencia se puede conseguir una drástica disminución del tamaño final del modelo mutado, mejorando el coste de la simulación.

Puesto que el caso general es el que se aplica a descripciones estructurales, el proceso de mutación se dividirá en dos fases:

1. Mutación del nivel principal del modelo.
2. Mutación de los componentes.

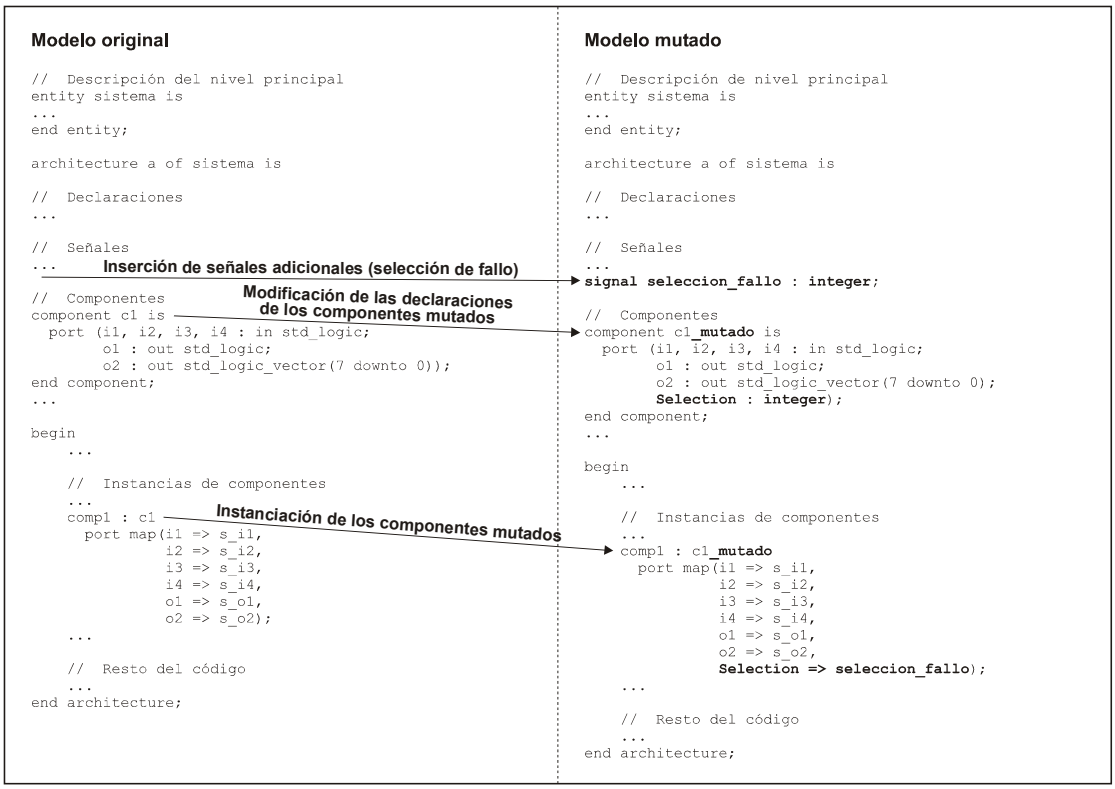

Figura 5.15: Generación automática de mutantes de un modelo. Mutación del modelo.

Evidentemente, y como ya se apuntó al analizar la inserción automática de perturbadores (véase el apartado 5.4.1.5), la tarea de generar mutaciones de forma automática no debe verse como una aplicación independiente, sino como una utilidad más de una herramienta de inyección de fallos. Para facilitar la tarea de modificar el código, será de gran ayuda contar con un

árbol lexicográfico del modelo, que permita la "sencilla" modificación de las sentencias (y otros elementos de código) existentes.

La Figura 5.15 muestra cómo se realizaría la mutación del nivel principal del modelo. En la figura se puede apreciar que la mutación afecta, por un lado, a la declaración e instanciación de los componentes, a los que se incorpora una entrada adicional llamada *Selection*, similar a la utilizada en los perturbadores. Su misión es indicar, con un valor numérico, cuál de las múltiples mutaciones que se pueden activar es la elegida para inyectar el fallo. Para poder llevar a cabo el control de la inyección es preciso declarar una señal interna adicional que, en tiempo de simulación, y utilizando la técnica de órdenes del simulador, se modificará para activar la mutación deseada.

Obsérvese la similitud del modo de controlar la inyección en el método expuesto con el de una de las versiones propuestas para la implementación de los perturbadores: la denominada optimizada (véase el apartado 5.4.1.6.1). Al igual que en dicho caso, con esta manera de controlar los mutantes en tiempo de simulación también es posible permite la inyección de fallos simples, o de múltiples en el tiempo sin solapamiento.

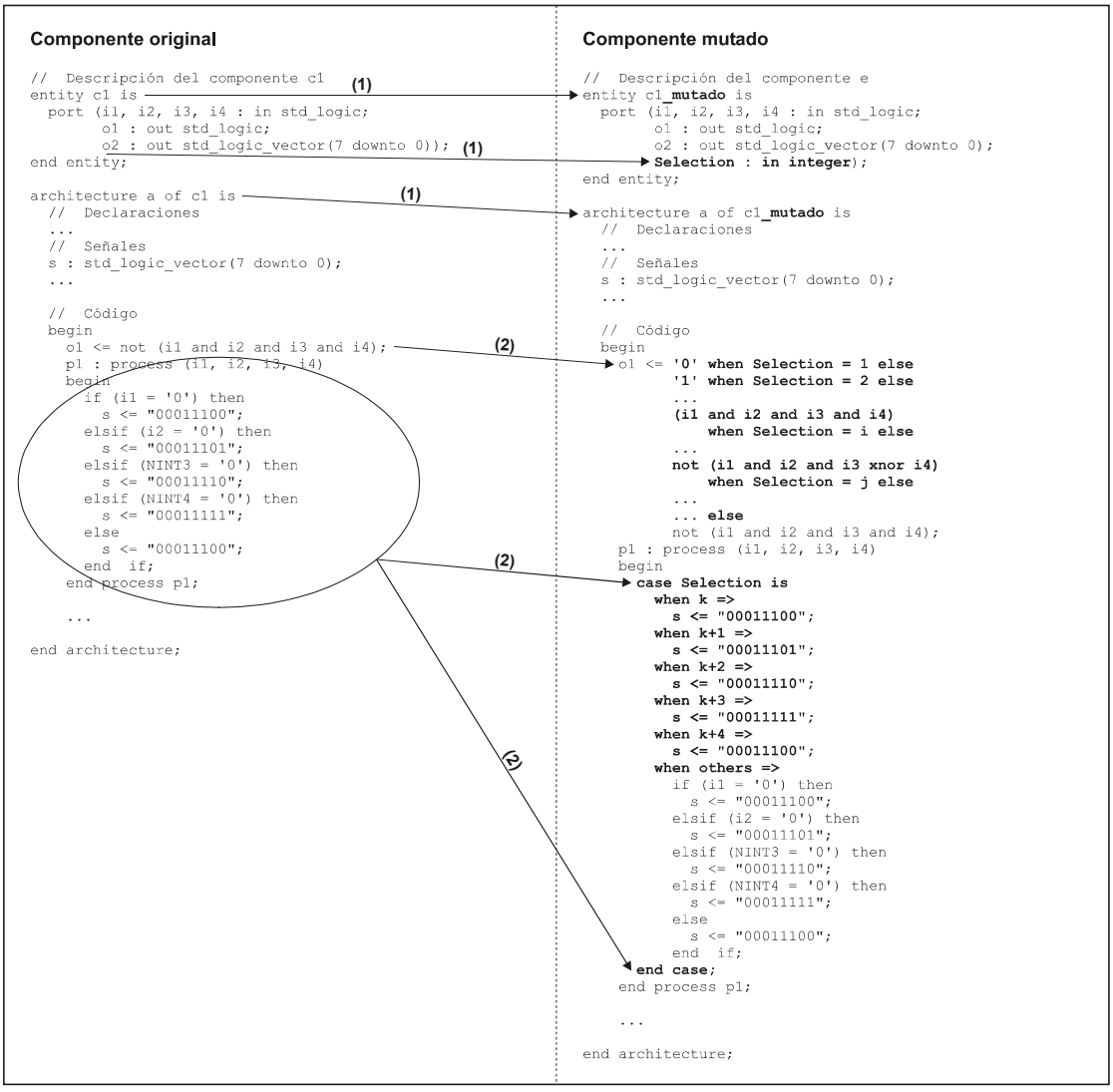

Figura 5.16: Generación automática de mutantes de un modelo. Mutación de componentes.

En cuanto a la mutación de la implementación de los componentes, ésta también se realiza en dos fases: la modificación de la interfaz (añadiéndole la entrada de selección que permite activar una determinada variación del código) y la alteración del código.

En el ejemplo mostrado en la Figura 5.16 se pueden apreciar estas dos vertientes. Con (1) se han etiquetado todas las variaciones relacionadas con la adaptación del componente: cambio del nombre de la entidad (porque no puede coincidir con el de la entidad sin mutar), adición de la entrada *Selection* a la entidad, y asociación de la arquitectura mutada (aunque tenga el mismo nombre) a la entidad mutada. Con la etiqueta (2) se han marcado algunas variaciones en el código de la arquitectura original.

Por ejemplo, la asignación de valor a la salida *o1* se ha transformado en una asignación de valor condicional, de forma que en función de diferentes valores de *Selection* se puede asignar un valor erróneo (valores fijos: modelos *stuck-at* '0', *stuck-at* '1', etc.; diversas funciones incorrectas para generar el nuevo valor, etc.) o el correcto (con el último `else`).

La sentencia `if` incluida en el proceso *p1* es muy similar a una sentencia `case`, dadas las múltiples opciones que se pueden dar. Para mutarla se ha introducido una sentencia `case`, de manera que en función de diferentes valores de *Selection* (empezando por el último empleado para mutar la sentencia de asignación anteriormente comentada), se pueden asignar cada una de las opciones del múltiple `if`, así como el valor correcto (el de la sentencia `if` sin modificar).

5.4.3 Otras técnicas

Además de las técnicas anteriores (órdenes del simulador, perturbadores y mutantes), en la bibliografía se han encontrado algunas variantes [DeLong *et al.* 1996, Sieh *et al.* 1997]. Su objetivo es atenuar los principales defectos de los perturbadores (exceso de señales de control) y los mutantes (necesidad de un elevado espacio de memoria para almacenar los componentes mutados, y el problema de la validez de los modelos de fallos en las descripciones comportamentales).

Estas técnicas se basan en independizar los modelos con y sin fallos. De esta manera se pueden utilizar los modelos existentes efectuando mínimos cambios en el código. Además, los diseñadores pueden usar el mismo modelo para simular el sistema sin fallos y con fallos, sin crear modelos separados o recompilar el modelo para cada fallo.

Sin embargo, para lograr este objetivo es necesario complicar la declaración de los tipos de datos, las funciones de resolución y las arquitecturas comportamentales, incluyendo simultáneamente los datos sin fallos, la máscara de fallos y el control de activación de los fallos. Además, hay que introducir compiladores y algoritmos de control ad hoc para manejar estas extensiones del lenguaje VHDL.

5.5 Comparación de las diferentes técnicas

La Tabla 5.4 resume a grandes rasgos las principales diferencias entre las técnicas más habituales, que son las implementadas en el presente trabajo: órdenes del simulador, perturbadores y mutantes.

		Ventajas	Inconvenientes
Técnica	**Órdenes del simulador**	Sencillez de implementación No es intrusiva (no requiere modificación del modelo) No introduce sobrecarga de espacio Sobrecarga de simulación independiente del modelo	Dependencia absoluta del simulador: • Modelos de fallos • Complejidad de los modelos
	Perturbadores	Inserción sencilla Reutilizabillidad de los perturbadores Poca sobrecarga de espacio Alta capacidad de modelado de fallos	Sobrecarga de simulación dependiente del número de perturbadores insertados Requieren modificación y recompilación del modelo
	Mutantes	Muy alta capacidad de modelado de fallos Reutilizabilidad de los mutantes	Sobrecargas de simulación y de espacio dependen del número de mutaciones introducidas y del tamaño del modelo Requieren modificación y recompilación del modelo

Tabla 5.4: Comparación de las técnicas más usuales de inyección de fallos mediante simulación de modelos en VHDL.

De la tabla se pueden extraer algunas conclusiones interesantes acerca de la utilidad y conveniencia de uso de cada una de las tres subtécnicas. Atendiendo a la completitud de los modelos de fallos:

- Cuando los modelos de fallos no necesitan ser muy complejos, la subtécnica más conveniente es la de *órdenes del simulador*. De todos modos, los simuladores comerciales son cada vez más potentes, y permiten inyectar un conjunto de modelos de fallo cada vez más completo.
- En cambio, para aplicar un conjunto elaborado de modelos de fallos, las subtécnicas con modificación del código parecen más apropiadas, dada su mayor capacidad de modelado. En particular, para descripciones comportamentales son más convenientes los *mutantes*, mientras que los *perturbadores* son más adecuados para descripciones estructurales.

Si se comparan las subtécnicas en función del coste, las *órdenes del simulador* resultan ser las que menos coste de elaboración del modelo sobre el que se inyectarán los fallos, debido su baja intrusividad (no requiere la modificación del modelo). Por el contrario, los *perturbadores* y *mutantes* requieren un coste (tanto espacial como temporal) muy elevado para la elaboración del modelo de inyección.

5.6 Ejemplos de herramientas y otras aportaciones

En este apartado se describen y comentan algunas de las publicaciones más destacables en relación con la inyección de fallos mediante simulación de modelos en VHDL. Entre estas publicaciones se encuentran tanto la descripción y/o aplicación de algunas herramientas importantes como la descripción de propuestas para optimizar las diferentes subtécnicas.

Un detalle importante que hay que destacar acerca de la bibliografía presentada es el hecho de que algunos de los trabajos expuestos pertenecen al ámbito del test. Sin embargo, las propuestas presentadas son lo suficientemente interesantes como para tenerlas en cuenta en el campo general de la inyección de fallos.

La exposición de los trabajos se realizará por orden cronológico, con independencia de la subtécnica particular utilizada.

- **MEFISTO** (*Multi level Error and Fault Injection Simulation TOol*) [Jenn *et al.* 1994] es la herramienta más emblemática (por su carácter precursor) de esta técnica de inyección. Fue desarrollada conjuntamente por el LAAS-CNRS de Toulouse (Francia) y la Universidad de Chalmers (Göteborg, Suecia). Su estructura se muestra en la Figura 5.17. En ella se pueden distinguir las tres fases en las que se divide un experimento de inyección: configuración (o *set-up*), simulación y análisis de resultados (o *data processing*).

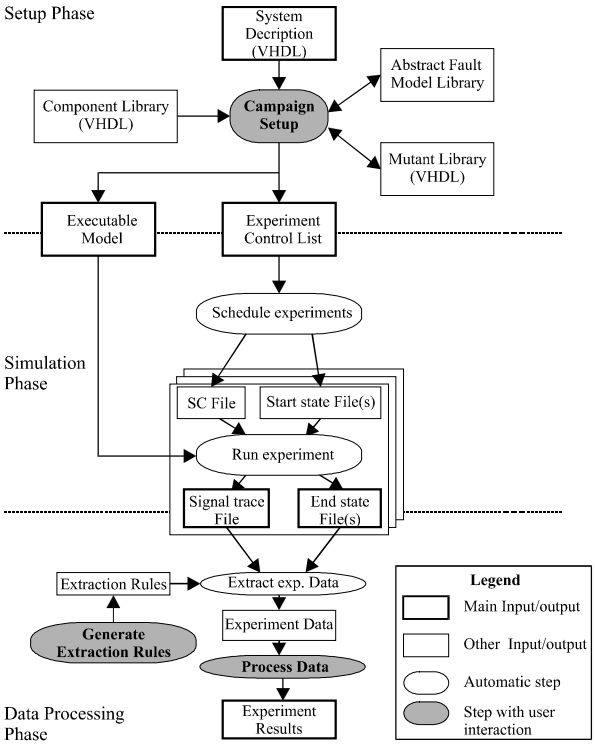

Figura 5.17: Estructura general de la herramienta MEFISTO [Jenn *et al.* 1994].

Funciona bajo UNIX sobre una red de estaciones de trabajo Sun IPC, lo cual permite realizar la inyección de los fallos de manera distribuida.

En cuanto a las técnicas que utiliza, en [Jenn *et al.* 1994] se asegura que permite inyectar fallos utilizando tanto *órdenes del simulador* como mutantes, pero no se ha encontrado ninguna publicación de la utilización de esta segunda técnica[60].

Con MEFISTO se pueden inyectar fallos transitorios y permanentes en señales y variables de modelos comportamentales y estructurales realizados al 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"[61]) o definidos por el usuario. Todos los modelos están definidos a partir del concepto de *modelos abstractos de fallos* (o AFMs, 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 los dos últimos casos, además, se puede establecer un número determinado de ocurrencias de la condición buscada.

En cualquier caso, el proceso de inyección sigue los siguientes pasos:

1. Detectar el instante de la inyección.
2. Detener el simulador.
3. Modificar los puntos del circuito afectados.
4. Reiniciar el simulador.

Posteriormente, los dos centros que realizaron MEFISTO desarrollaron sendas herramientas basadas en ésta. Así surgieron MEFISTO-C (realizada en la Universidad de Chalmers) y MEFISTO-L (desarrollada en el LAAS-CNRS). Ambas herramientas tienen características completamente diferentes, por lo que se tratarán por separado.

- En [DeLong *et al.* 1994, Ghosh *et al.* 1995b, DeLong *et al.* 1996] se presenta un método para inyectar fallos tanto en modelos comportamentales como estructurales. Aunque conceptualmente es similar a los perturbadores, implica la modificación de tipos de datos y funciones de resolución, por que quedaría enclavado en la categoría "otras técnicas".

  Los elementos encargados de afectar al correcto funcionamiento del sistema se denominan controladores de la inyección de fallos, o FIC (del inglés *fault injection controller*)[62] [DeLong *et al.* 1996]. Estos FIC son elementos que se insertan sobre el modelo original con la misión de alterar el valor de una señal.

  En función de si el modelo es estructural o comportamental (e independientemente del nivel de abstracción), los FIC son de diferente clase. Así, en diseños estructurales un FIC es un componente que se interpone, bien entre dos componentes o entre una línea externa (de entrada, salida o bidireccional) y un componente. En diseños comporta-

[60] De hecho, en [Leveugle y Hadjiat 2000] se asegura que no ha sido realmente implementada.

[61] Bajo esta denominación se incluyen 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.

[62] Inicialmente, en [DeLong *et al.* 1994, Ghosh *et al*. 1995b] los denominaban módulos de inyección de fallos, o FIM (del inglés *fault injection module*).

mentales, en cambio, se implementa como un proceso que afecta a una determinada señal.

Para este segundo caso, se proponen una serie de modificaciones adicionales en el código, entre las que se encuentra un nuevo tipo de dato para la señal que se verá afectada, y una función de resolución para resolver la colisión provocada por el proceso FIC. El nuevo tipo de dato introducido es un registro (`record`) que maneja información como el valor que se asigna y la máscara de fallo que se inyecta. Mediante esta máscara es posible inyectar tanto fallos simples como múltiples.

Los modelos de fallos que se pueden inyectar dependen del nivel de abstracción del modelo del sistema, y pueden ir del clásico *stuck-at* ('0', '1') para los modelos realizados en los niveles lógico y de bloque funcional[63] hasta los más complejos en modelos de alto nivel, como por ejemplo la corrupción de paquetes en un sistema de comunicaciones. Por corrupción de paquetes se entiende eliminación o duplicación de paquetes enteros, alteración de bits, etc.

La forma de operar de ambos tipos de FIC es similar. A través de un fichero leen las características de cada fallo que se ha de inyectar (no se especifica quién ni cómo lo genera): instante de inyección, duración del fallo, lugar de inyección y máscara de inyección.

Este método (en sus dos modalidades) se ha aplicado a varios modelos, entre los que se incluyen:

- Un modelo a nivel ISA[64] de un ICS (del inglés *interlocking control system*), que es un sistema utilizado para el control del tráfico ferroviario. Está basado en el microprocesador MC6809 de Motorola. Sobre él se han inyectado fallos *stuck-at* ('0', '1') permanentes y transitorios.
- Diferentes versiones, realizadas a distintos niveles de abstracción (según la notación de Walker-Thomas), de un monitor de guardia, o WMC (del inglés *watchdog monitor card*).
  - En el modelo a nivel algorítmico se han inyectado fallos en sus entradas, tanto en los datos como en los códigos de operación. Los fallos inyectados en los datos son *stuck-at* ('0', '1') tanto simples como múltiples. En los códigos de operación, han consistido en la transformación a otro código de operación válido, pero incorrecto.
  - En el modelo a nivel de bloque funcional se han inyectado fallos *stuck-at* ('0', '1') en algunas señales internas del modelo.

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 [Kumar *et al.* 1994, Ghosh *et al.* 1995a], una herramienta de inyección de fallos mediante simulación descrita en el capítulo 3, más concretamente en el apartado 3.2.5.

---

[63] En estos trabajos se utiliza la notación de Walker-Thomas [Walker y Thomas 1985]. Los niveles lógico y de bloque funcional se corresponden con los niveles de puerta y RT utilizados en el presente capítulo.

[64] Un modelo en el nivel ISA (del inglés *Instruction Set Architecture*) es un modelo comportamental capaz de ejecutar código máquina real (es, pues, similar al RT).

- En [Celeiro *et al.* 1996] se presenta otro modelo de perturbador para implementar fallos *bridge* en modelos estructurales al nivel de puerta. En este caso, el modelo perturbado incorpora todos los posibles perturbadores. De esta manera, con la activación adecuada de las entradas de control de los diferentes perturbadores, se pueden inyectar fallos simples y/o múltiples.
- En [Amendola *et al.* 1996] se presenta una contribución referida al uso de perturbadores. Los experimentos expuestos muestran la validación de los mecanismos de tolerancia a fallos de un sistema utilizado para la gestión de las comunicaciones en una línea de metro. En concreto, se trata de una placa controladora MVME162 de Motorola. Para ello, sobre el modelo a nivel RT del sistema (no se especifica si es estructural o comportamental) se añadieron tres módulos encargados de la inyección de fallos (en el *bus*, los registros de la CPU y la memoria) y otro para el control de la inyección, encargado de generar el instante, el lugar de inyección y el manejo de los perturbadores. El modelo de fallo utilizado es el *bit-flip*.
- En el campo del test, uno de los objetivos principales es la reducción del número de patrones de test que hay que aplicar a un sistema sin que se reduzcan la validez estadística de los resultados obtenidos. Este objetivo se denomina *Fault List Collapsing*, o simplemente *Fault Collapsing*. [Smith *et al.* 1996, Aftabjahani y Navabi 1997, Benso *et al.* 1998a, Benso *et al.* 1999a, Parrotta *et al.* 2000, Berrojo *et al.* 2002] son algunos de los trabajos (todos ellos orientados a la tarea de test) en los que se trata esta problemática, sea para la inyección de fallos en VHDL o para otras técnicas de inyección[65].

  En particular, en [Aftabjahani y Navabi 1997] se aplica la técnica de mutantes sobre modelos de puertas lógicas (los modelos de fallos son *stuck-at* (‘0’, ‘1’)). En [Parrotta *et al.* 2000] se utiliza la técnica de órdenes del simulador, combinada con puntos de ruptura, para inyectar fallos transitorios de tipo *bit-flip* sobre el modelo de un microprocesador i8051 ejecutando distintos programas de prueba (*benchmarks*). Por su parte, en [Berrojo *et al.* 2002] 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 (*Solar Array Drive Electronics*), fabricado por Alcatel Espacio.
- **VERIFY** (*VHDL-based Evaluation of Reliability by Injecting Faults efficientlY*) [Sieh *et al.* 1997] es una herramienta perteneciente a las clasificadas como “otras técnicas” (véase el apartado 5.4.3). Utiliza unas señales especiales de inyección denominadas FIS (*Fault Injection Signals*), que se declaran en la arquitectura comportamental de los componentes y que identifican el tipo de fallo, el tiempo medio entre fallos y su duración media.

  Se han desarrollado un compilador para manejar las extensiones del lenguaje, y un simulador para ejecutar los experimentos de inyección. El compilador extrae las señales FIS y se las suministra al simulador, que de esta forma tiene acceso a los parámetros de

[65] En efecto, [Benso *et al.* 1999a] es un trabajo enclavado en la inyección mediante SWIFI. Sin embargo, la eliminación de inyecciones que *a priori* sean redundantes, o no obtengan ningún dato relevante es un tema que por su interés merece ser incluido en este capítulo. Obviamente, el objetivo de esta reducción de las campañas es la reducción del tiempo de simulación sin perder validez estadística.

los fallos. Para iniciar un experimento, el usuario especifica el tiempo de simulación y el número de fallos que deben inyectarse. Cada experimento consiste en una ejecución sin fallos (*golden run*) y *k* ejecuciones con fallos. Cada vez que se debe inyectar un fallo, el simulador determina automáticamente (de acuerdo con las descripciones del modelo) el tipo, lugar, instante y duración del fallo. Las trazas de las simulaciones con y sin fallos se comparan con la herramienta TRACEDIFF, que analiza la propagación de los errores, la probabilidad de avería o recuperación del sistema, y otros parámetros de la Confiabilidad del sistema. En la Figura 5.18 se puede observar el diagrama de bloques de VERIFY.

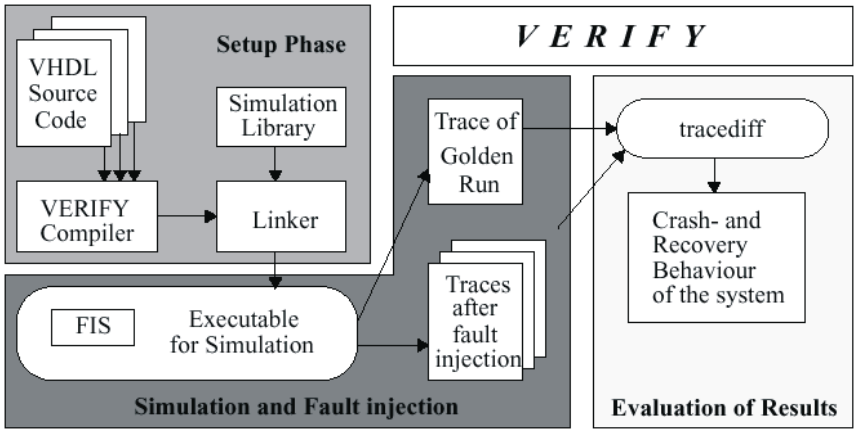

Figura 5.18: Estructura general de la herramienta VERIFY [Sieh *et al.* 1997].

En [Sieh *et al.* 1997] se presenta la aplicación de VERIFY sobre una versión reducida del procesador DP32, al que se inyectan fallos *stuck-at* (‘0’, ‘1’) (en los *pines* y en la circuitería combinacional) y *bit-flip* (en los registros internos).

- De **MEFISTO-C** [Folkesson *et al.* 1998] se conocen muy pocos detalles. Al igual que su predecesora:
  - ⇨ Se ejecuta sobre una red de estaciones de trabajo Sun Sparc bajo UNIX, por lo que también permite inyectar fallos de manera distribuida.
  - ⇨ Se puede aplicar a modelos realizados a nivel de puerta y RT.
  - ⇨ Utiliza exclusivamente la técnica de órdenes del simulador.
  - ⇨ Permite inyectar fallos transitorios y permanentes, y los modelos de fallos son los mismos que en MEFISTO.

  En cuanto a las pocas diferencias que se pueden destacar con respecto a MEFISTO, se puede decir que la generación del instante de inyección se realiza uniformemente en el tiempo.

- **MEFISTO-L** [Boué *et al.* 1998], en cambio, está muy bien documentada. En la Figura 5.19 se muestra su estructura general. En ella se puede distinguir a grosso modo las tres fases de la inyección, como ya se explicó para MEFISTO.

  En este caso, la evolución con respecto a MEFISTO ha sido radical, ya que la única técnica de inyección que permite utilizar son los *perturbadores*. Los modelos de fallo que se pueden aplicar pueden ser predefinidos (como los modelos *stuck-at* (‘0’, ‘1’), *bit-flip*,

indetermination y *open-line*, asociados a los tipos "bit" –véase [61]) o definibles por el usuario, al igual que sucedía con su antecesora. Los perturbadores predefinidos están almacenados en una biblioteca de perturbadores. Como característica peculiar de los perturbadores en MEFISTO-L (sean predefinidos o de usuario), cabe destacar que cada perturbador sólo implementa un modelo de fallo.

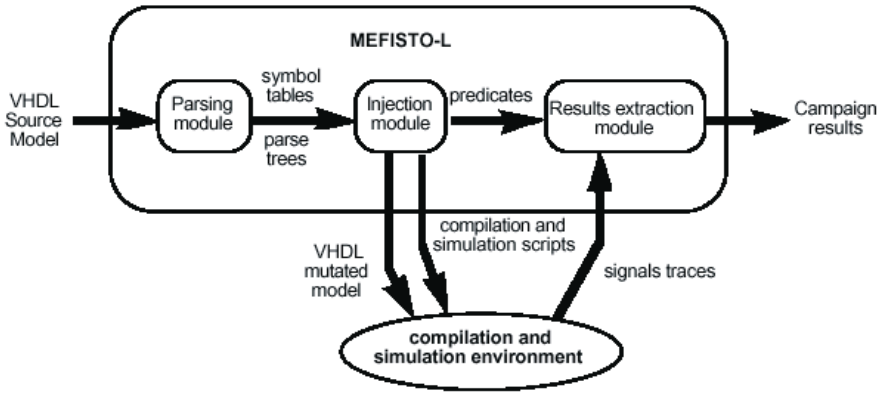

Figura 5.19: Estructura general de la herramienta MEFISTO-L [Boué *et al.* 1998].

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, *Injection module*) o para la observación de la evolución del circuito tras la inyección de fallos.

En cuanto a las similitudes que mantiene con MEFISTO, merece la pena resaltar la realización de la inyección de forma distribuida, ya que se ejecuta sobre una red de estaciones de trabajo sobre UNIX, los niveles de abstracción de los modelos a los que se puede aplicar (niveles de puerta y RT), los tipos de fallos que se pueden inyectar (transitorios y permanentes) y las formas de generar el instante de inyección (utilización de valores de tiempo y de patrones de disparo).

- Otro campo de trabajo en el que también se utilizan las técnicas de inyección de fallos sobre modelos en VHDL es la inyección mediante emulación con FPGA. La técnica más frecuentemente utilizada es la de mutantes (sobre descripciones a 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 [Leveugle y Hadjiat 2000, Leveugle y Hadjiat 2002].
  - ⇨ Modificando la descripción al nivel de puerta del diseño. En particular, los biestables del modelo se reemplazan por otros que son, a su vez, un diseño estructural que

implementa un mecanismo similar al *Scan-Chain* (véase el apartado 3.3.2.3) [Velazco *et al.* 2001, Civera *et al.* 2001c].

- En [Vargas *et al.* 2000] se utiliza inyección de fallos mediante simulación en VHDL para validar modelos de sistemas en los que se han introducido mecanismos de detección y corrección de errores (EDAC, del inglés *error detection and correction*) mediante una herramienta llamada FT-PRO. Para ello se emplea un mecanismo similar a los mutantes.

  El modelo original es modificado por FT-PRO para dotarle, por un lado, de mecanismos EDAC (consistentes en códigos Hamming y bits de paridad), así como de un proceso capaz de inyectar fallos (de tipo *bit-flip*). Los instantes de inyección y duración de los fallos se generan siguiendo un criterio predefinido para la tasa de errores, o MTBF (del inglés *mean time between failures*).

- **Fault Detector System** [Corno *et al.* 2000] es un prototipo de herramienta que aplica la técnica de *órdenes del simulador* a modelos de nivel RT. La herramienta funciona sobre una estación de trabajo Sun Ultra 5 bajo UNIX, y su diagrama de bloques se muestra en la Figura 5.20.

  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'.

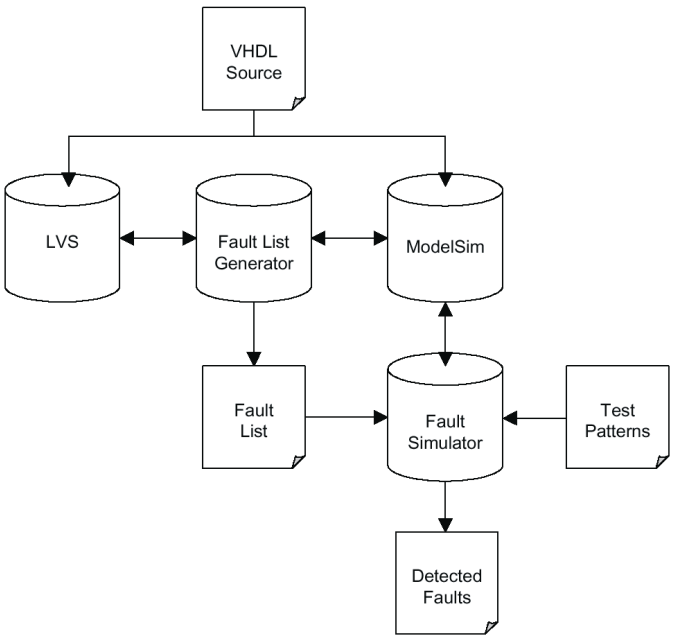

Figura 5.20: Diagrama de bloques de la herramienta Fault Detector System [Corno *et al.* 2000].

- En [Shaw *et al.* 2001] se presenta un modelo de perturbador para inyectar fallos de tipo *bridge* en modelos estructurales al 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 de 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 (cuyo efecto puede ser por ejemplo una alteración de los retardos).

- En [Cardarilli *et al.* 2002] se muestra una implementación peculiar de perturbadores. La peculiaridad estriba en que se aplican a un diseño comportamental. Esto es así porque el perturbador no es un componente, sino un proceso adicional que se añade al código comportamental, y que 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.

5.7 Resumen y conclusiones

En este capítulo se han analizado los diferentes métodos existentes para la inyección de fallos mediante la simulación de modelos en VHDL: la utilización de órdenes del simulador, perturbadores, mutantes, y otras técnicas que requieren la modificación de la sintaxis del lenguaje.

En primer lugar, se han descrito las características que han hecho del lenguaje de descripción de *hardware* VHDL uno de los más empleados, tanto para modelar sistemas en cualquier nivel de abstracción, como para la inyección de fallos.

Dada la orientación del presente trabajo de tesis, se ha incidido especialmente en las tres primeras: órdenes del simulador, perturbadores y mutantes. Para cada método se han descrito con detalle la forma de operar, las aproximaciones más interesantes y los modelos de fallos que permiten inyectar. Además, para las técnicas de perturbadores y mutantes, se han propuesto nuevos modelos, orientados a su integración en una herramienta general de inyección de fallos.

En lo que se refiere a los modelos de perturbadores propuestos, se ha presentado un amplio espectro de versiones, de diferente complejidad, y por consiguiente, con distintas características generales. Estas características se basan en dos aspectos primordiales: (a) las clases de fallos que se pueden inyectar atendiendo a su multiplicidad, y (b) el impacto que su integración produce en el modelo. Es evidente que, cuanto mayor es el número de señales que hay que añadir al modelo original para generar el modelo perturbado, éste resulta más complejo, y deriva en una ralentización del proceso de simulación.

A la luz del conjunto de modelos de perturbadores propuestos, se ha visto que es preciso alcanzar un compromiso entre las capacidades de inyección deseadas y la complejidad del modelo resultante.

En cuanto a los mutantes, el tema es más espinoso, porque la técnica presenta de por sí un inconveniente importante: un coste espacial muy elevado a causa de las múltiples modificaciones (mutaciones) potenciales del código del modelo. Además, entre las propuestas existentes no hay ninguna que permita la inyección de fallos transitorios sin problemas. Laa principales dificultades existentes radican en la sincronización de las simulaciones con y sin fallos y en el elevado coste temporal asociado al cambio de estado.

Se ha propuesto un nuevo método que resuelve el problema de la sincronización de las versiones con y sin fallos. Sin embargo, tiene el inconveniente de que es una solución "por fuerza bruta", puesto que se basa en realizar una única versión modificada de los ficheros del modelo, en la que se incluyan todas las mutaciones que se pretenda modelar (bien se trate de todas las posibles o aquéllas que han sido previamente seleccionadas). Para la inyección de los fallos, en tiempo de simulación basta con seleccionar la mutación deseada.

Por último, se enumeran y comentan los trabajos publicados más relevantes en relación con la utilización del lenguaje VHDL para inyectar fallos, aunque no pertenezcan expresamente a la simulación de modelos en VHDL. Así, se han incluido trabajos relacionados con el test, con la emulación de fallos mediante FPGA, etc. En estos trabajos se analizan tanto las herramientas de inyección de fallos más significativas, como propuestas de modelos de fallos y técnicas no automatizadas para la inyección de fallos sobre modelos en VHDL empleando cualquiera de sus modalidades.

5.8 Trabajo futuro

Los modelos de perturbadores y mutantes propuestos se encuentran en fase de prueba sobre modelos de sistemas simples. Para verificar su correcto funcionamiento deben aplicarse a otros modelos de sistemas más complejos, como los descritos en el capítulo 7.

Asimismo, también es preciso completar la automatización de la inserción de los perturbadores en el modelo y de la generación automática de las mutaciones del modelo.

En el caso particular de la propuesta de implementación de la técnica basada en mutantes, además, hay dos aspectos todavía pendientes: la representatividad de los modelos de fallos y la optimización del tiempo de simulación.

En cuanto a la representatividad de los modelos de fallos, es conveniente comprobar la equivalencia entre los diferentes modelos de fallos propuestos y los fallos en el nivel físico, y seleccionar el conjunto de modelos que mejor representen a éstos.

Con respecto a la optimización del tiempo de simulación, podría ser interesante buscar nuevas alternativas o variantes de implementación que solventen los problemas temporales de las técnicas más clásicas, y que no perjudiquen el coste espacial.

6 La herramienta de inyección de fallos VFIT

En este capítulo se describe la herramienta de inyección de fallos desarrollada (VFIT: VHDL-based Fault Injection Tool). En primer lugar se hace referencia a un prototipo previo desarrollado por el Grupo de Sistemas Tolerantes a Fallos, al que el autor pertenece. En el siguiente apartado se definen algunos términos utilizados durante la descripción de la herramienta. Después se describe la herramienta en sí, comenzando por una exposición de sus características generales, para posteriormente mostrar y explicar su diagrama de bloques. Por último, se presenta un informe del estado actual de la herramienta, así como algunas ideas para futuras incorporaciones o modificaciones.

6.1 Antecedentes

La herramienta desarrollada parte de un prototipo realizado en el seno del Grupo de Sistemas Tolerantes a Fallos (GSTF) de la Universidad Politécnica de Valencia, cuya estructura se muestra en la Figura 6.1 [Gil *et al.* 1998a, Gil 1999].

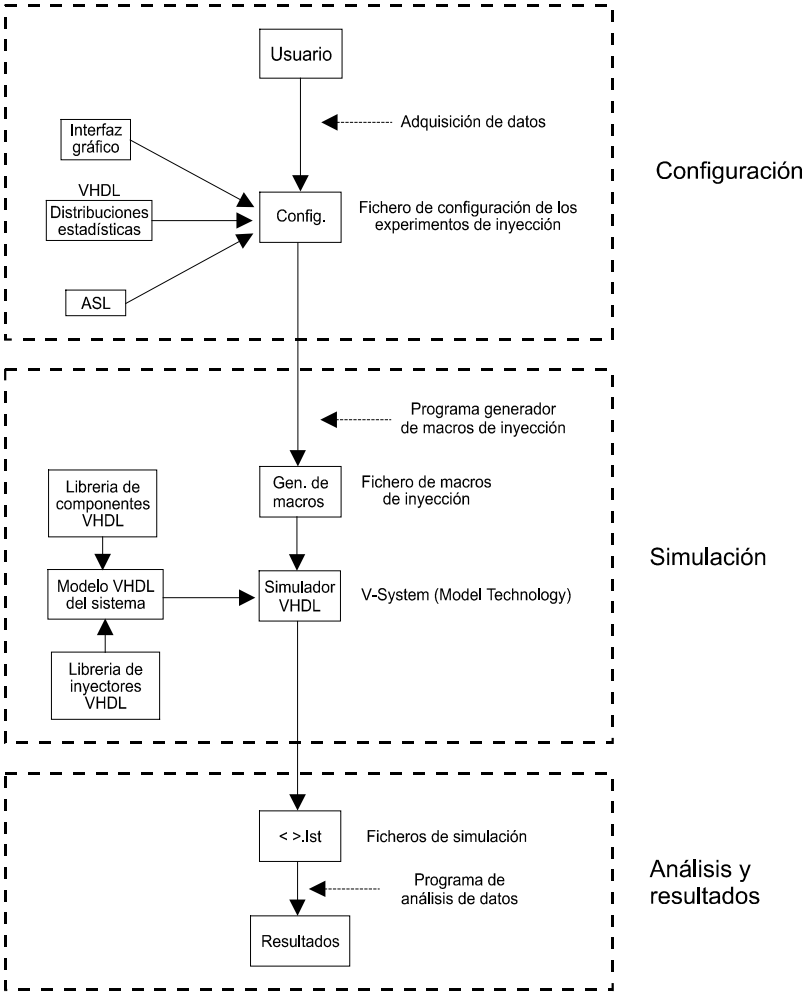

Figura 6.1: Estructura de la herramienta original.

El objetivo principal era la implementación de un inyector de fallos mediante simulación de modelos en VHDL que funcionara sobre un PC en entorno Windows.

Estaba pensada para inyectar fallos en modelos en VHDL a nivel de chip, incluyendo los niveles lógico, de registro y algorítmico (en descripciones comportamentales). El diseño del prototipo se realizó en torno al simulador comercial V-System de Model Technology [ModelTech 1997].

La técnica de inyección empleada en el primer prototipo fue la basada en la utilización de las órdenes del simulador.

Se trataba de una herramienta sencilla, adecuada para llevar a cabo campañas de inyección sobre modelos de complejidad baja/media. Con ella se han obtenido resultados sobre la validación de sistemas microcomputadores, tanto en el estudio del síndrome de errores [Gil *et al.* 1997b, Gil *et al.* 1997c, Gil *et al.* 1998b, Gil *et al.* 1998c, Gil *et al.* 1998d], como en la validación de los mecanismos de detección y recuperación de errores [Gil *et al.* 1999, Gil *et al.* 2000, Gracia *et al.* 2000].

La estructura de la herramienta mostrada en la Figura 6.1 se basa en las fases de operación que se siguen para la realización de un experimento de inyección: *configuración*, *simulación* y *análisis y extracción de resultados*.

6.1.1 Fases de la inyección de fallos

6.1.1.1 Configuración

En esta primera fase se definen los parámetros de la campaña de inyección de fallos, almacenándolos en un fichero de configuración. Los parámetros más importantes son:

- El número de inyecciones.
- El instante de inyección.
- La duración del fallo.
- El lugar de la inyección.
- El tipo de fallo inyectado.

6.1.1.2 Simulación

Consiste en simular el modelo VHDL a partir del fichero generado en la fase de configuración, haciéndolo dos veces: una sin inyectar fallos, y otra inyectando los fallos determinados. Esta segunda simulación se efectúa utilizando una *macro* de inyección que se genera a partir del fichero de configuración. Como resultado se generan una serie de ficheros de traza: uno con la simulación del modelo sin fallos (llamada *golden run*) y uno más por cada una de las inyecciones realizadas.

6.1.1.3 Análisis y extracción de resultados

En esta fase se lleva cabo la comparación de la traza de la simulación sin fallos con las de las simulaciones con fallos. La comparación contempla, además de las diferencias de valor en los diferentes puntos del modelo, los siguientes aspectos:

- Las señales relacionadas con los mecanismos de detección de errores y tolerancia a fallos.
- Las cláusulas[66] de propagación/detección/recuperación de errores. En caso de que se cumpla alguna, se almacena el instante de simulación correspondiente.

A partir de los datos almacenados (discordancia de valores, activación de las señales de detección de errores, etc.) se realiza un tratamiento de los datos, con el fin de obtener medidas relativas al síndrome de error o a la validación de los mecanismos de detección y recuperación de errores:

- Clasificación de los fallos y errores.
- Latencias de propagación, detección y recuperación.
- Coberturas de detección y recuperación.

6.1.2 Inconvenientes del prototipo original

Este primer prototipo presentaba algunas deficiencias o problemas que convenía subsanar. Las más significativas eran:

- La elaboración de la lista de posibles objetivos de inyección no se realizaba de manera automática. Para cada modelo, con la ayuda de un *parser* sintáctico, se generaba una lista de señales y variables atómicas susceptibles de ser alteradas.
- La interactividad con el usuario era alta. La herramienta estaba implementada casi íntegramente en VHDL, por lo que su ejecución se realizaba desde el simulador. Debido a esto, la ejecución de las tres fases de la inyección se realizaba de forma independiente, y cada una de las fases tenía que ser activada manualmente por el usuario.
- Sólo utilizaba la técnica de inyección mediante órdenes del simulador.

Además de éstas, tenía otras carencias causadas por las características de los equipos y de la versión del simulador utilizada, entre las que cabe destacar:

- El simulador tenía problemas de gestión de memoria, lo cual impedía hacer campañas con un número elevado de inyecciones, así como la simulación de modelos de complejidad media.
- La herramienta de compilación y simulación estaba integrada en un entorno rígido que impedía la ejecución separada de sus distintas utilidades. Este hecho afectaba a la autonomía de la herramienta de inyección durante los procesos de inyección y análisis de resultados.

[66] Una cláusula es una ecuación booleana que expresa una condición del comportamiento del modelo.

- No era posible aplicar algunos tipos de fallo. De hecho, sólo se inyectaban fallos *stuck-at* (‘0’, ‘1’), *indetermination* y *open-line*.

En cualquier caso, como herramienta preliminar de inyección de fallos sobre modelos VHDL resultó muy válida, dando lugar a un conjunto de estudios (tanto del síndrome de error como de validación) que se comentarán en el capítulo 7. Por este motivo se decidió la realización de un nuevo prototipo, basado en el anterior, que solucionara la mayor parte de los problemas encontrados.

6.2 Características

A partir del prototipo original en VHDL, se ha desarrollado una nueva versión [Baraza 1999] (a la que se ha denominado VFIT: VHDL-based Fault Injection Tool) cuyas características veremos clasificadas en generales y técnicas.

6.2.1 Características generales

1. Funciona sobre un PC en entorno Windows®.
2. Es **General**, aplicable a cualquier modelo VHDL.
3. Es totalmente **Automática**, ya que existe una única aplicación (fichero ejecutable) realizada en C++ desde la que se llevan a cabo todas las fases de la inyección mostradas en la Figura 6.1.
4. Es fácilmente utilizable por el usuario.

A continuación se explica qué métodos se han seguido para cumplir algunos de los objetivos anteriormente expuestos.

6.2.1.1 Generalidad

La consecución de este objetivo se ha abordado desde dos puntos de vista muy diferentes, si bien en algunos casos, complementarios:

1. Tratar todos los modelos del mismo modo, extrayendo unas características “comunes”.
2. Objetivar los parámetros necesarios para la inyección.

6.2.1.1.1 Extracción de características comunes

Para poder llevar a cabo esta tarea, ha sido preciso implementar una utilidad llamada Analizador Sintáctico y Lexicográfico (ASL). Su tarea es rastrear el conjunto de ficheros que componen un modelo en VHDL, localizar en ellos las declaraciones de señales y variables, y generar un árbol lexicográfico del modelo. Posteriormente, en función de la técnica de inyección que se vaya a utilizar, buscará los elementos donde se puede inyectar, y se los mostrará al usuario para que seleccione, de entre todos los lugares posibles, un subconjunto de posibles puntos de inyección en la campaña.

Además, para poder implementar un elevado número de modelos de fallo en las técnicas de utilización de las órdenes del simulador y en los perturbadores, el ASL también investiga los tipos de dato de los *ports*, señales y variables en los que se puede inyectar, hasta reducirlos a su expresión básica (su tipo base). De este modo, VFIT será capaz de mostrar al usuario un

gran rango de modelos de fallo que se pueden inyectar, para que elija cuáles quiere realmente utilizar.

En cuanto a la técnica de los mutantes, aunque todavía no está integrada en VFIT, el ASL es capaz de localizar los puntos del código donde se puede realizar una mutación.

6.2.1.1.2 Objetivación de los parámetros

Para lograr este aspecto, se ha establecido una potente interfaz inyector-usuario, de manera que éste pueda fijar de manera objetiva los aspectos dependientes del sistema y del método de inyección que el usuario quiere utilizar. De este modo, la herramienta procesa esta información de manera general.

La introducción de esta "comunicación" entre el programa y el usuario se ha efectuado utilizando las capacidades gráficas del lenguaje C++, y el compilador utilizado ha sido el Builder® C++ 3 de Borland®.

Grosso modo, los parámetros que la aplicación necesita se pueden clasificar en:

- Parámetros del experimento: técnica utilizada, número de fallos inyectados, idioma en que se generan los resultados (castellano/inglés), etc.
- Parámetros temporales (referentes al sistema o al experimento de inyección): duración del ciclo de reloj, instante de inyección, duración de la inyección, tiempo de simulación del experimento, etc.
- Parámetros de configuración del sistema. En diseños VHDL complejos, en los que hay procesadores, memorias, etc., para poder simularlos es necesaria la existencia de ficheros (*macros*) de configuración, para cargar algo (programas y/o datos) en memoria, ejecutar rutinas de inicio en procesadores, etc.
- Selección del o los lugares donde se va a inyectar, así como el o los modelos de fallos que el usuario desea inyectar en cada lugar.
- Parámetros dependientes del sistema. Para poder evaluar el comportamiento del sistema ante fallos, es preciso que el usuario especifique qué elementos del diseño (señales o variables) están relacionados con la detección y/o recuperación de errores, así como las cláusulas de detección y/o recuperación y las señales donde se detecta la propagación de errores.

Para poder objetivar estos parámetros, en particular los que se refieren **directamente** al diseño (la selección del lugar de la inyección y los tipos de fallos en cada lugar), es necesario un preprocesamiento del modelo para seleccionar, en función de la técnica utilizada, los candidatos para sufrir una inyección. Este preprocesamiento lo lleva a cabo, como se comentó en el apartado 6.2.1.1, el Analizador Sintáctico y Lexicográfico.

Una vez establecidos todos los parámetros necesarios, la herramienta es capaz de realizar la campaña solicitada:

1. Simulando el comportamiento del sistema con las condiciones generales indicadas, pero sin inyectar fallos. De este modo, el programa obtiene la respuesta normal del sistema, que constituye la simulación de referencia (llamada *golden run*).
2. Simulando el comportamiento del sistema, bajo las mismas condiciones generales, e inyectando el número de fallos que se le ha indicado.

3. Comparando los datos obtenidos en cada una de las inyecciones con los de la simulación sin fallos, para efectuar el análisis indicado (estudio del síndrome de error o validación), y generando automáticamente los resultados.

6.2.1.2 Automatización

Este objetivo es el más simple y, al mismo tiempo, el más complejo de todos. Es preciso realizar un programa que, mediante un sistema de menús más o menos complejo, permita recibir, ordenar y clasificar los parámetros necesarios para hacer una inyección de fallos sobre un modelo dado. Esa es la parte simple. La complejidad reside en lo que subyace por debajo de la herramienta: el simulador de VHDL en el que se basa, que es un programa comercial, **independiente** de la herramienta de inyección.

Si se pretende hacer una herramienta totalmente automática, como es el caso que nos ocupa, desde el mismo programa se debería ejecutar en segundo plano el simulador (y en algunos casos, también el compilador). Con la versión del simulador utilizada en la herramienta original (V-System) esto era muy complicado, por ser un programa integrado en un entorno general desde el que se ejecutaban sus diferentes utilidades. Sin embargo, con la nueva versión (ModelSim [ModelTech 2001a, ModelTech 2001b]), se puede hacer de manera más sencilla, aunque ello añade nuevas complicaciones a la hora de realizar el programa:

- Configuración. La herramienta de inyección debe ser capaz de conocer la configuración del simulador, para poder acceder a datos necesarios para su trabajo. Así mismo, debe poder configurar al simulador para la realización de las campañas de inyección, indicándole las librerías utilizadas (nombre y ubicación), así como otros muchos parámetros.
- Sincronización. La herramienta debe poder conocer el estado del simulador en todo momento, así como interpretar el resultado de la ejecución de la utilidad invocada (esto es, el resultado de una compilación o simulación).
- Gestión. Para poder utilizar el simulador, es preciso disponer de **licencia**, que se valida mediante un programa gestor de licencias. Se utiliza el programa FLEX*lm* (*FLEX license manager* [Globetrotter 1998], de Globetrotter Software Inc., distribuido junto con ModelSim, que también puede ejecutarse en segundo plano. La herramienta debe ser capaz de ejecutar el gestor de licencias para ver si puede continuar la ejecución o no.

6.2.2 Especificaciones técnicas

Las principales características técnicas de VFIT son:

1. Se puede aplicar a cualquier modelo en VHDL, estructural o comportamental, independientemente del nivel de abstracción (puerta, lógico, chip, etc. –véase el apartado 4.1) utilizado, y de su complejidad.
2. Implementa todas las técnicas de inyección de fallos sobre modelos en VHDL: órdenes del simulador, perturbadores y mutantes[67].

[67] En el prototipo actual sólo se permite la inyección de fallos mediante órdenes del simulador, si bien las otras dos técnicas han sido probadas [Gracia *et al.* 2000, Gracia *et al.* 2001a, Gracia *et al.* 2001b] pero no han sido integradas en la herramienta.

3. Permite inyectar un elevado número de modelos de fallos, aparte de los clásicos *stuck-at* (‘0’, ‘1’) y *bit-flip* (ver apartado 6.3.5).
4. Se pueden inyectar fallos simples: permanentes, transitorios o intermitentes.
5. Genera resultados en formatos estándar (tablas en formatos Excel o Gnuplot, gráficas, etc.) para que sean fácilmente interpretables[68].

6.3 Estructura

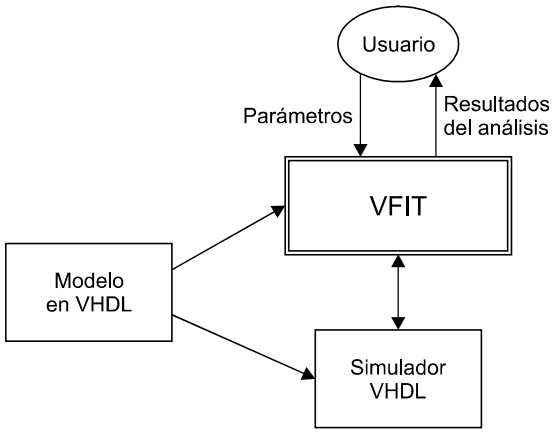

Figura 6.2: Estructura general de la nueva herramienta.

Con los aspectos comentados hasta ahora, la herramienta se podría representar, de manera muy general, mediante el diagrama de bloques de la Figura 6.2. En él se puede ver cómo, a partir de

1. la interacción con el usuario para el establecimiento de los parámetros del experimento,
2. el modelo en VHDL del sistema que se desea validar, y
3. el simulador VHDL,

la herramienta es capaz de devolver al usuario los resultados del análisis del experimento.

Este diagrama de bloques es muy general, ya que oculta los aspectos más importantes de la herramienta, descritos anteriormente: su generalidad y automatización. Por ello, en el diagrama de bloques de la Figura 6.3 se muestra más en detalle el interior de la herramienta.

A la vista de la figura, se pueden extraer dos diferencias fundamentales respecto al prototipo original:

1. Todos los elementos están **integrados** en una aplicación única y centralizada, desde la que se activan por medio de menús.
2. El simulador es invocado desde la herramienta, como si fuera un elemento más de la misma. En el prototipo preliminar, el simulador era el componente fundamental de la herramienta, pues prácticamente todos los elementos estaban implementados en VHDL. En la figura, para resaltar la independencia del simulador respecto a la herra-

[68] Esta característica todavía no ha sido implementada. En la actualidad, los resultados se presentan en forma de tablas en ASCII, fácilmente importables desde otras aplicaciones.

mienta, se le ha representado de manera diferente a los elementos que sí pertenecen a VFIT.

En el diagrama de bloques se pueden ver sus elementos, que guardan cierta similitud con los del prototipo preliminar, ya que la estructura subyacente es la misma, si bien sus funciones y contenidos son más complejos. Además se incorpora el bloque denominado Configuración de la herramienta, cuya misión es la sincronización y configuración entre la herramienta y el simulador.

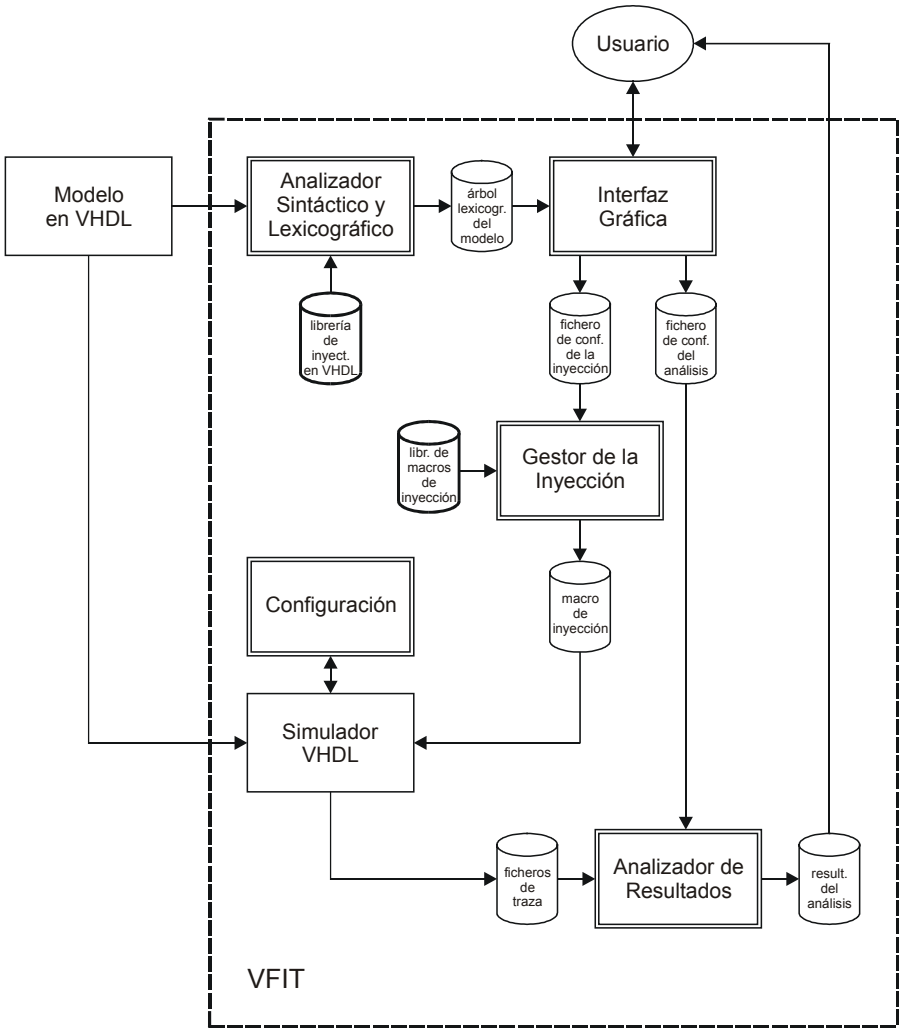

Figura 6.3: Estructura detallada de VFIT.

Otra de las diferencias existentes con la primera versión de la herramienta es el hecho de que, en lugar de haber un fichero de configuración, hay dos: uno con los parámetros referentes a la inyección, y otro con los referentes al análisis de resultados. El primero prácticamente coincide con el de la herramienta preliminar, si bien por el proceso de generalización de la herramienta, el número de parámetros es muy superior. El segundo contiene básicamente los nombres de los elementos del modelo que indican la detección y la recuperación de errores, y las cláusulas correspondientes. En la versión preliminar, las cláusulas estaban integradas en el propio programa de análisis, lo que afectaba a la generalidad del prototipo.

En los próximos apartados se describen los diferentes elementos de VFIT.

6.3.1 Librería de inyectores en VHDL

Esta librería contiene ficheros en VHDL con modelos predefinidos de perturbadores para su inserción en el modelo cuando se utilice esta técnica de inyección.

6.3.2 ASL-Árbol lexicográfico del modelo

El ASL es una utilidad que, como ya se ha comentado, genera a partir del modelo en VHDL del sistema sobre el que se va a inyectar, un árbol lexicográfico con todos los elementos del modelo, en función de la técnica de inyección que se vaya a utilizar. Este árbol contiene una relación jerarquizada de los elementos del modelo sobre los que se puede realizar una inyección. Así,

- para la inyección basada en órdenes del simulador contiene los *ports*, señales y variables,
- para la inyección mediante perturbadores, los *ports* y las señales utilizadas para conectar componentes, y
- para la inyección mediante mutantes, todos los elementos sintácticos del código del modelo que se pueden alterar.

6.3.3 Interfaz Gráfica-Ficheros de configuración

Es un completo sistema de menús y ventanas gráficas a través de las cuales el usuario puede especificar todos los parámetros necesarios para llevar a cabo un experimento de inyección de fallos sobre modelos en VHDL utilizando cualquiera de las técnicas posibles: órdenes del simulador, perturbadores y mutantes.

Con este módulo se puede especificar:

- El tipo de inyección.
- Los parámetros de la inyección: temporales, lugar(es) de inyección, tipos de fallos posibles en cada lugar, etc.
- El tipo de análisis (estudio del síndrome de error o validación), y los elementos del modelo (señales y/o variables) que indican la detección y recuperación de errores, así como las cláusulas correspondientes.

Como ya se ha indicado anteriormente, este módulo genera dos ficheros de configuración: uno con los parámetros de la inyección y otro con la configuración del análisis.

6.3.3.1 Parámetros de inyección

Se pueden destacar los siguientes:

- La técnica utilizada.
- El número de fallos inyectados.
- Los posibles lugares donde se puede inyectar un fallo, seleccionando entre los *ports*, señales y variables del árbol lexicográfico del modelo.

- Los modelos de fallo que se pueden inyectar para cada posible lugar de inyección. Estos se dividen en dos clases, en función de la técnica de inyección utilizada.

  En el caso de las técnicas de *órdenes del simulador* y *perturbadores*, los modelos de fallo que se pueden aplicar dependen del tipo de dato al que pertenece. Para este menester, VFIT maneja todos los tipos de datos de forma general, atendiendo a su categoría según la sintaxis del VHDL. Así, todos los tipos datos que se pueden utilizar en un modelo son de algunos de estos tipos:

  ⇨ Escalares (o simples):

  - Enumerados, como los tipos digitales (`bit`, `std_logic`, etc.), booleanos, descripciones de estados de un máquina de estados, etc.
  - Numéricos, entre los que se incluyen todos los tipos y subtipos enteros (basados en el tipo estándar `integer`) y reales (basados en el tipo estándar `real`).
  - Físicos, similares a los numéricos pero con unidades.

  ⇨ Compuestos:

  - Vectores de cualquier tipo base.
  - Registros (`record`) con campos de cualquier clase, excepto punteros.

  Esta clasificación de los tipos de datos es similar a la realizada en [Riesgo y Uceda 1996] (véase el apartado 5.4.2), con la salvedad de que VFIT sí maneja los datos de tipo `record`, ya que al fin y al cabo no son más que colecciones de las otras clases.

  Los modelos de fallos que se pueden inyectar con VFIT se agrupan atendiendo a esta misma clasificación. En particular, sólo se consideran los elementos básicos, ya que los datos de tipos compuestos se pueden ver, en último término, como conjuntos de elementos de tipos simples. Así, los modelos de fallos que VFIT permite aplicar son:

  ⇨ En tipos enumerados:

  - Fijar a un valor cualquiera de la lista.
  - Conmutar entre dos valores cualesquiera.

  ⇨ En tipos numéricos y físicos:

  - Fijar a un valor del rango de valores (incluyendo las unidades en tipos físicos), sea especificado directamente o generado aleatoriamente.

  Desde el punto de vista de los tipos digitales, se pueden inyectar los modelos indicados en la Tabla 6.1, tal como se describieron en el capítulo 4.

  En el caso de los *mutantes*, los modelos de fallos se refieren a cambios sintácticos en el código VHDL, en particular a los ocho modelos especificados en [Armstrong *et al.* 1992], indicados en el apartado 5.4.2: *stuck-then*, *stuck-else*, *assignment control*, *dead process*, *dead clause*, *microoperation*, *local stuck-data* y *global stuck-data*.

- El nombre y la duración del programa de prueba, $t_{workload}$. Este programa establece los estímulos del modelo en la simulación.
- La duración de la simulación, t_{simul}.

Técnica de inyección		Tipos de fallos: Transitorios	Tipos de fallos: Permanentes e Intermitentes
	Órdenes del simulador	*Bit-flip*, *Pulse*, *Indetermination*, *Delay*	*Stuck-at* ('0', '1'), *Indetermination*, *Open-line*, *Delay*
	Perturbadores	*Bit-flip*, *Pulse*, *Indetermination*, *Delay*	*Stuck-at* ('0', '1'), *Indetermination*, *Open-line*, *Delay*, *Stuck-open*, *Short*, *Bridging*

Tabla 6.1: Modelos de fallos que se pueden inyectar con VFIT en elementos con tipos de datos digitales.

- El instante de inyección, t_{iny}. Se puede especificar un instante fijo o generarlo con una función de distribución. Actualmente están implementadas algunas funciones típicas relacionadas con la ocurrencia de fallos (transitorios, permanentes e intermitentes): Uniforme, Exponencial, Normal y Weibull (véase el apéndice A).
- La duración de cada fallo, Δt_{iny}. VFIT permite inyectar fallos permanentes, transitorios e intermitentes. En el caso de los fallos transitorios, se puede establecer una duración fija o generarla aleatoriamente con las funciones de distribución indicadas para el instante de inyección. Para fallos intermitentes, hay que definir tres parámetros: la duración de la actividad (tiempo que el fallo está activo), la duración de la inactividad (el tiempo entre dos activaciones) y el tamaño de la ráfaga (número de activaciones). Todos ellos se pueden fijar o generar aleatoriamente.

6.3.3.2 Configuración del análisis

Aquí se especifica si el análisis realizado consistirá en el estudio del síndrome de error o en la validación de un sistema tolerante a fallos.

Para un estudio del síndrome de error, hay que clasificar los fallos y los errores, y establecer (mediante cláusulas) las condiciones bajo las cuales se determina cuándo se ha propagado un fallo y se ha producido un error de los determinados en la clasificación de errores. Por defecto, VFIT es capaz de discernir si un fallo ha sido efectivo o no, tanto por si ha sido sobrescrito o por si permanece latente. En los parámetros habrá que especificar los nombres de las diferentes clases de errores efectivos, así como las condiciones que las determinan.

Para la validación de un sistema tolerante a fallos, es preciso especificar los diferentes mecanismos de detección y recuperación de errores del modelo, así como las cláusulas que determinan cuándo el modelo ha detectado/recuperado un error.

6.3.4 Librería de *macros* de inyección

Esta librería consta de un conjunto de ficheros que contienen *macros* predefinidas, escritas en el lenguaje de órdenes del simulador, que se utilizarán para la inyección de fallos.

6.3.5 Gestor de la Inyección-*Macro* de inyección

Esta utilidad controla el proceso de inyección. Para ello, partiendo de los parámetros de inyección genera *n*+1 *macros*: una para simular el modelo sin fallos, y *n* con invocaciones a las *macros* de inyección, siendo *n* el número de fallos que hay que inyectar.

Una vez generada la *macro*, invoca al simulador para que la ejecute, generando como salida *n*+1 ficheros de traza: uno con la simulación del modelo sin fallos (*golden run*), y *n* con un fallo inyectado.

La salida del Generador de *macros* es una *macro* comprensible por el simulador VHDL. Como se indicó anteriormente, hay dos tipos de *macro*: una que simula el modelo sin fallos y otra para su simulación con la inyección de un fallo. El contenido de la *macro* de inyección para la simulación con fallos es, a su vez, un conjunto de llamadas a unas *macros* patrón, que realizan la inyección de fallos sobre elementos del modelo.

Estas *macros* patrón forman parte de la librería de *macros* de inyección, y se pueden invocar con parámetros. Éstos dependen de los parámetros de inyección especificados en el fichero de configuración de la inyección, y entre ellos se puede destacar: el lugar de la inyección (que dependiendo de la técnica de inyección utilizada puede ser una constante genérica –`generic`–, un puerto –`port`–, una señal –`signal`–, una variable, un elemento sintáctico –sentencia `if`, asignación de señal o variable, sentencia `case`, etc.–), el formato de representación de los valores en las trazas (binario, hexadecimal, etc.), el modelo de fallo, el tipo y duración del fallo, etc.

De esta manera se optimiza espacio, ya que la *macro* de inyección sólo contendrá, aparte de la configuración de la inyección, la llamada a la *macro* patrón utilizada con sus parámetros correspondientes.

Los modelos de fallos que se pueden aplicar con VFIT son los especificados en el apartado 6.3.3.1.

6.3.6 Analizador de Resultados

Esta utilidad se encarga de comparar la traza de la simulación sin fallos con las de las simulaciones con fallos. La información generada depende del tipo de análisis realizado:

- En caso del estudio del síndrome de error, se indican (en función de las clasificaciones de fallos y errores especificadas en los parámetros) los porcentajes de errores efectivos (propagados), así como sus latencias.

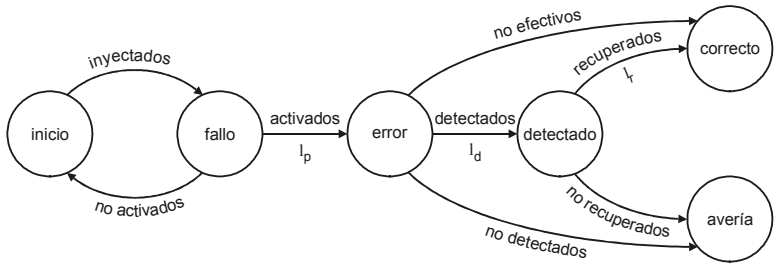

Figura 6.4: Grafo de predicados de los mecanismos de tolerancia a fallos.

- En la validación de un sistema tolerante a fallos, los resultados pueden utilizarse para cumplimentar el grafo de predicados de los mecanismos de tolerancia a fallos, mostrado en la Figura 6.4. Este grafo refleja la patología de los fallos desde que se inyectan hasta que los errores producidos son detectados y, ocasionalmente, recuperados. Del grafo se pueden obtener fácilmente las coberturas de detección y recuperación, así como las latencias medias de propagación, detección y recuperación.

En el apartado 6.3.6.1 se describen en detalle los diferentes parámetros numéricos calculados por el Analizador de resultados.

En la actualidad, la salida generada es un conjunto de ficheros en ASCII que contienen la información especificada en forma de tablas, aunque como ya se indicó en el apartado 6.2, está previsto cambiar el formato de salida para que devuelva los resultados de la inyección en un formato estándar: tablas en formatos Excel o Gnuplot, gráficas, etc.

6.3.6.1 Parámetros generados en el análisis

A continuación se especifican y describen los parámetros calculados por VFIT en los diferentes tipos de análisis [Baraza *et al.* 2002]:

- **Porcentaje de errores activados** (P_A). Se define mediante la ecuación (6.1)

$$P_A = \frac{N_{Activados}}{N_{Inyectados}} = \frac{N_{Activados}}{n} \tag{6.1}$$

  donde n es el número de fallos inyectados y $N_{Activados}$ es el número de errores activados. Un error se considera activado cuando es originado por un fallo activado (un fallo que produce un cambio en alguna señal o variable del modelo) que se propaga a las *señales de propagación* del modelo. Estas señales registran la propagación de los fallos y su manifestación como error, y dependen de la arquitectura del sistema bajo estudio.

- Cobertura de detección de errores (C_d). En realidad se han definido dos estimadores de esta cobertura:
  - ⇨ **Cobertura de los mecanismos de detección** ($C_{d(mecanismos)}$), definida mediante la ecuación (6.2)

$$C_{d(mecanismos)} = \frac{N_{Detectados}}{N_{Activados}} \tag{6.2}$$

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

  - ⇨ **Cobertura de detección global del sistema** ($C_{d(sistema)}$), definida mediante la ecuación (6.3)

$$C_{d(sistema)} = \frac{N_{Detectados} + N_{No\ efectivos}}{N_{Activados}} \tag{6.3}$$

    en la que $N_{No\ efectivos}$ representa el número de errores no efectivos. Un error no efectivo no afecta al resultado de la aplicación (carga de trabajo, o *workload*) ejecutada por el sistema. Se puede producir cuando la información errónea es sobrescrita por la ejecución normal del sistema, o porque permanece latente en una parte del siste-

ma no utilizada. En el último caso, el error podría ocasionalmente convertirse en efectivo. La no efectividad de los errores está relacionada con la redundancia intrínseca del sistema. Esta es la razón por la que se define esta cobertura más global.

- Cobertura de recuperación de errores (C_r). También se han definido dos estimadores:
  - ⇨ **Cobertura de los mecanismos de recuperación** ($C_{r(mecanismos)}$), definida por la ecuación (6.4)

$$C_{r(mecanismos)} = \frac{N_{Detectados\_recuperados}}{N_{Activados}} \tag{6.4}$$

siendo $N_{Detectados\_recuperados}$ el número de errores detectados por los mecanismos de detección y recuperados por los mecanismos de recuperación.

  - ⇨ **Cobertura de recuperación global del sistema** ($C_{r(sistema)}$), definida a través de la ecuación (6.5)

$$C_{r(sistema)} = \frac{N_{Detectados\_recuperados} + N_{No\ efectivos}}{N_{Activados}} \tag{6.5}$$

donde $N_{no\ efectivos}$ tiene el mismo significado que para $C_{d(sistema)}$.

- **Latencia de propagación** (L_p). Se calcula mediante la ecuación (6.6)

$$L_p = t_p - t_{iny} \tag{6.6}$$

en la que t_p es el instante en que el error llega a las señales o variables de propagación, (es decir, cuando el error se activa), y t_{iny} es el instante de inyección.

- **Latencia de detección** (L_d). Se calcula con la ecuación (6.7)

$$L_d = t_d - t_p \tag{6.7}$$

donde t_d es el instante en que el error activado es detectado por los mecanismos de detección.

- **Latencia de recuperación** (L_r). Se define mediante la ecuación (6.8)

$$L_r = t_r - t_d \tag{6.8}$$

siendo t_r el instante en que el error detectado es recuperado por los mecanismos de recuperación.

De los parámetros enumerados, P_A y L_p se calculan tanto en el análisis del síndrome de error como en la validación de un sistema tolerante a fallos. Los demás (coberturas y latencias de detección y recuperación), que implican a los mecanismos de detección y recuperación del sistema, se calculan exclusivamente durante una validación.

6.3.7 Configuración

Como ya se explicó en el apartado 6.2.1.2, este módulo tiene por misiones:

- Leer la configuración del simulador (en particular, las librerías utilizadas). Con esta información, el ASL actualizará una biblioteca interna de constantes, funciones de resolución y tipos de datos que utilizará para la generación del árbol lexicográfico del modelo.
- Configurar el simulador, a partir de los parámetros de inyección especificados mediante la Interfaz gráfica, para la realización de los experimentos de inyección. Parámetros configurables son: librerías utilizadas por el modelo, directorio de trabajo del modelo, opciones de compilación y/o simulación, etc.

6.4 Optimización del tiempo de simulación

El problema principal de la inyección de fallos mediante simulación radica en el tiempo de simulación. VFIT no escapa a este problema, y se hace patente cuando tanto el modelo sobre el que se va a inyectar como la carga de trabajo (*workload*) que ejecuta el modelo son complejos. Evidentemente, al utilizar un simulador comercial, la duración de la simulación depende en gran medida del simulador utilizado, así como de las características del ordenador sobre el que se ejecuta.

Para poder solventar este problema, se ha dotado a VFIT de la **posibilidad** de dividir un experimento de inyección en partes independientes, denominadas **sesiones**. De las tres fases en que se divide un experimento (véase el apartado 6.1.1), la primera es la especificación de los parámetros, y las otras dos (*simulación* y *análisis y extracción de resultados*) son "operativas". Una sesión es una parte de estas fases operativas. Ejemplos de sesiones son:

a) La simulación de todas las inyecciones, analizando las trazas generadas. Esta es la sesión más completa, que coincide con el experimento.

b) La simulación de todas las inyecciones, sin analizar las trazas generadas.

c) La simulación de una parte de las inyecciones, realizando un análisis (parcial) de las trazas generadas.

d) La simulación de una parte de las inyecciones, sin analizar las trazas generadas.

e) (Sin inyección previa) El análisis de las trazas generadas por el total de las inyecciones.

f) (Sin inyección previa) La recopilación de los análisis parciales existentes para llevar a cabo el análisis global.

La partición de la simulación conlleva un incremento en la complejidad de la fase de análisis y generación de resultados, debido a la necesidad de efectuar un proceso final de fusión de los resultados.

Gracias al mecanismo de partición, es posible llevar a cabo un experimento con elevado coste temporal realizando sesiones en paralelo en diferentes ordenadores, reduciendo notablemente el efecto del tiempo de simulación. En la actualidad, la planificación y ejecución de las sesiones se deja al usuario, si bien está previsto que sea VFIT quien las realice de manera distribuida, suponiendo la existencia de una red de equipos capaces de realizar alguna de las clases de sesiones.

6.5 Ejemplos de aplicación de VFIT

A continuación se muestran dos ejemplos de aplicación de VFIT, correspondientes a dos de los experimentos incluidos en el capítulo 7. En cada uno de ellos se indican cómo se realiza la configuración, la *macro* de inyección generada y el resultado del análisis. En el apartado 6.5.1 se presenta una de las campañas dedicadas al estudio del síndrome de error del procesador PIC, descrito en el apartado 7.2.3, y cuyos resultados se comentan en el apartado 7.5.1. En el apartado 6.5.2 se muestra uno de los experimentos de inyección realizados para la validación del sistema computador tolerante a fallos basado en el procesador MARK2 descrito en el apartado 7.2.2, y cuyos resultados se comentan en el apartado 7.4.1.

Dado que para ilustrar el ejemplo se muestran algunos de los menús de VFIT, se remite al lector al manual de la herramienta [VFIT 2002] para una mayor aclaración respecto al sistema de menús de la herramienta.

6.5.1 Ejemplo de análisis del síndrome de error

Como ya se ha explicado, en este apartado se muestran los pasos que hay que seguir para llevar a cabo un experimento de inyección orientado al estudio del síndrome de error de un sistema no tolerante a fallos.

El experimento se realiza sobre el modelo en VHDL del microcontrolador PIC (descrito en el apartado 7.2.3). En los próximos apartados se describen los parámetros del experimento, divididos en dos tipos: los referidos a la inyección y los referidos al análisis. Después se muestra una fragmento de la *macro* de inyección generada para la realización de la inyección. Por último se exponen los ficheros de resultados obtenidos tras el análisis.

6.5.1.1 Parámetros de la inyección

Los parámetros correspondientes a la inyección son:

1. **Número de fallos inyectados**: 3000.
2. **Período de la señal de reloj (T)**: 100 ns.
3. **Carga de trabajo**: Algoritmo de ordenación de la burbuja (bubblesort), para n = 10 enteros.
4. **Duración de la carga de trabajo (T_W)**: 80 μs.
5. **Duración de la simulación**[69] **(T_S)**: 82 μs.
6. **Lugares de inyección**: Se pueden inyectar fallos en todas las señales combinacionales de la ALU y en la señal de reloj.
7. **Instante de inyección**: Se genera de manera aleatoria, siguiendo una distribución Uniforme entre 0.0 ns y T_W (T_W = 80 μs).

[69] Generalmente, la duración real de la simulación incluye tanto la duración de la carga de trabajo como un tiempo de observación posterior a la finalización de la carga, durante el cual el sistema todavía puede reaccionar ante los fallos insertados.

8. **Duración de los fallos**: Se inyectan fallos transitorios, cuya duración sigue una distribución Uniforme entre 0.1 T y 1.0 T, donde T es el período de la señal de reloj, especificado en 2.
9. **Modelos de fallos**: Se van a inyectar fallos *pulse*, *indetermination* y *delay*.

El modo en que estos parámetros se introducen en VFIT por parte del usuario se muestra en las ventanas que aparecen desde la Figura 6.5 a la Figura 6.12.

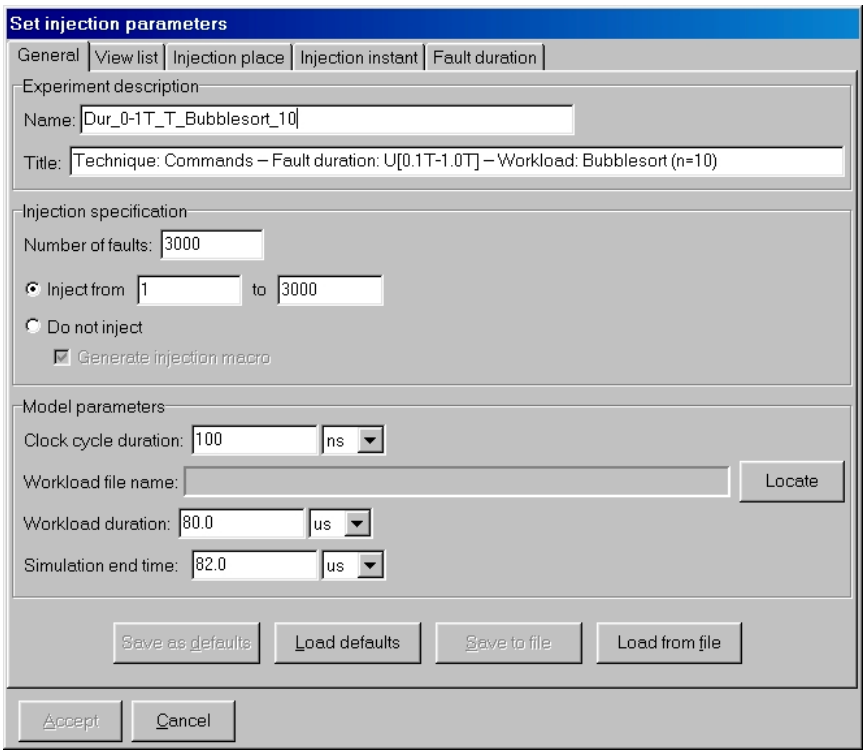

Figura 6.5: Especificación del número de fallos inyectados, la duración del ciclo de reloj, la carga de trabajo (implícita) y las duraciones de la carga y de la simulación completa.

Obsérvese que, a diferencia de la configuración mostrada en la Figura 6.16, en la de la Figura 6.5 no se especifica explícitamente la carga de trabajo mediante un fichero (el campo Workload file name está vacío). Esto se debe a que en el modelo correspondiente a este ejemplo, la carga está incluida de forma implícita en uno de los ficheros fuente del diseño.

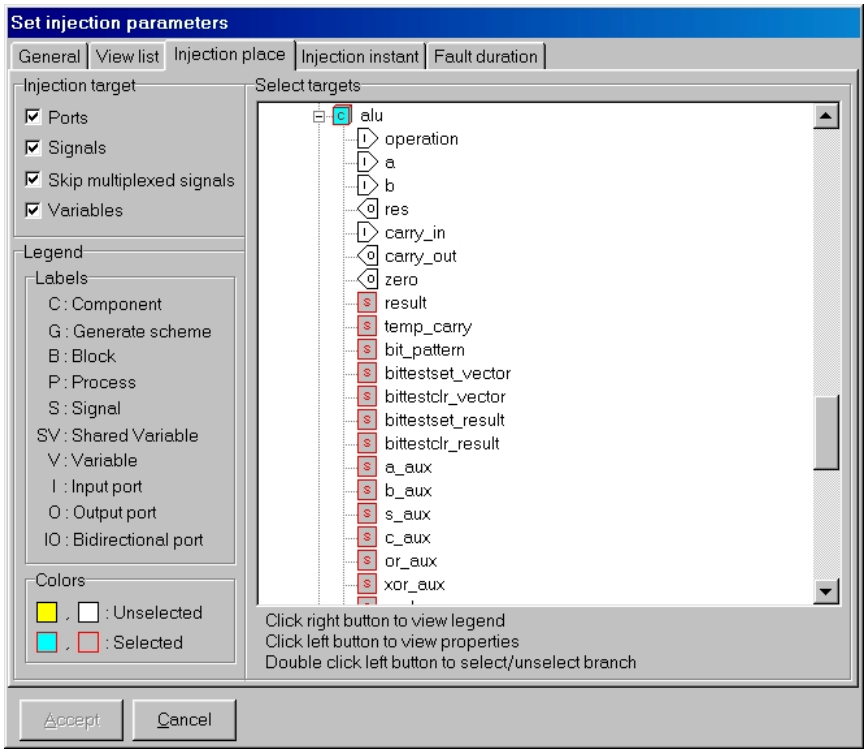

Figura 6.6: Selección de los lugares de inyección.

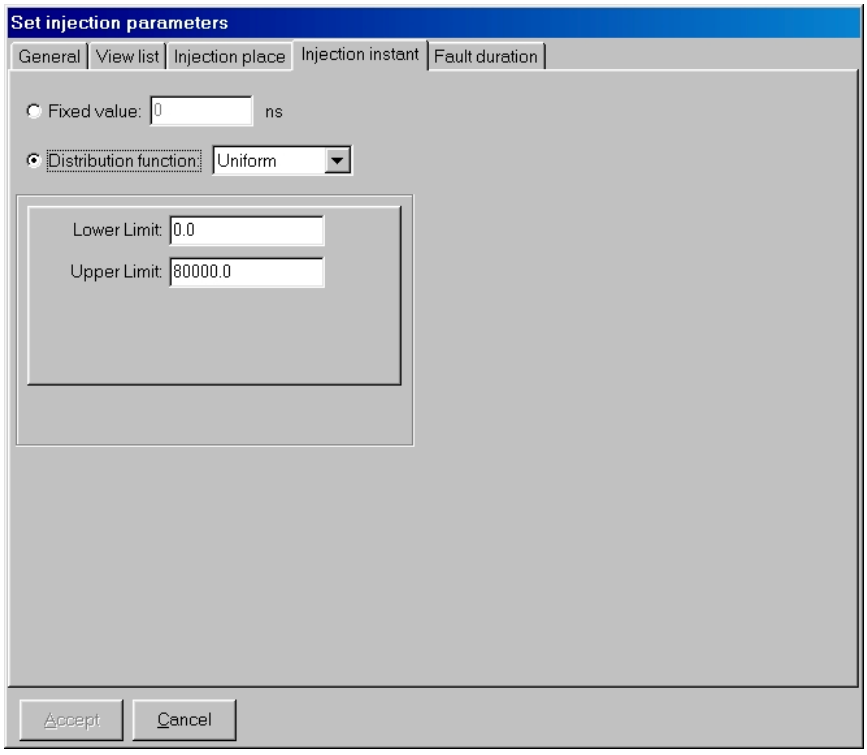

Figura 6.7: Especificación del instante de inyección.

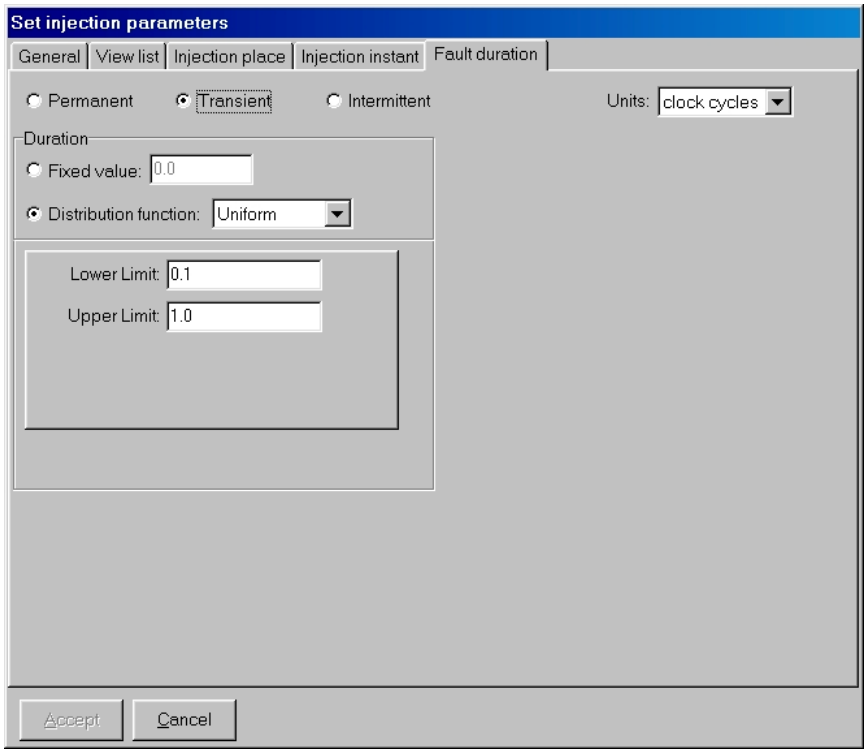

Figura 6.8: Especificación de la duración de los fallos.

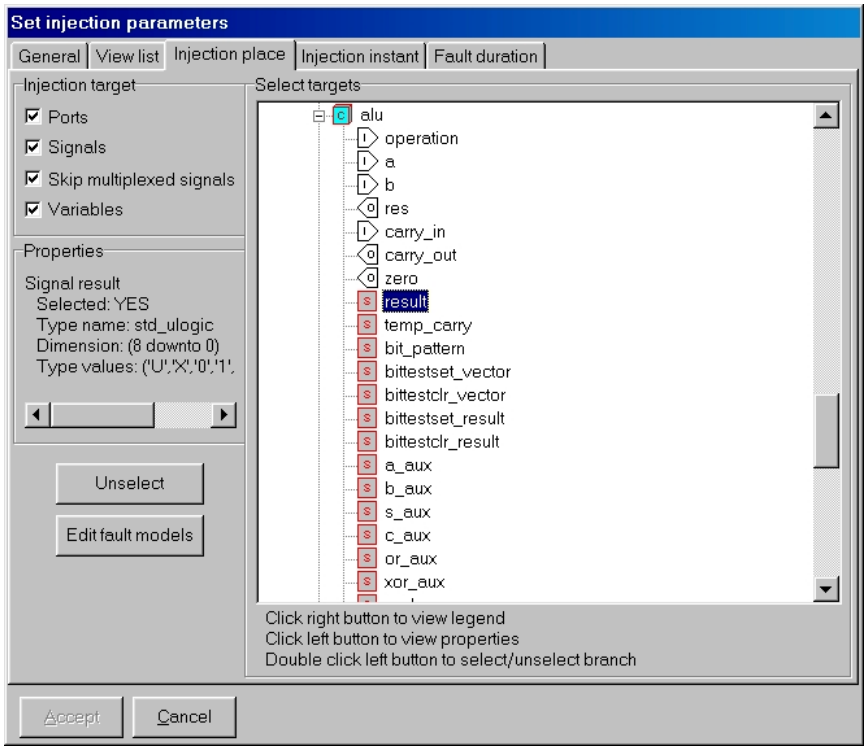

Figura 6.9: Especificación de los modelos de fallos *pulse* e *indetermination*. Paso 1: elección del lugar de inyección.

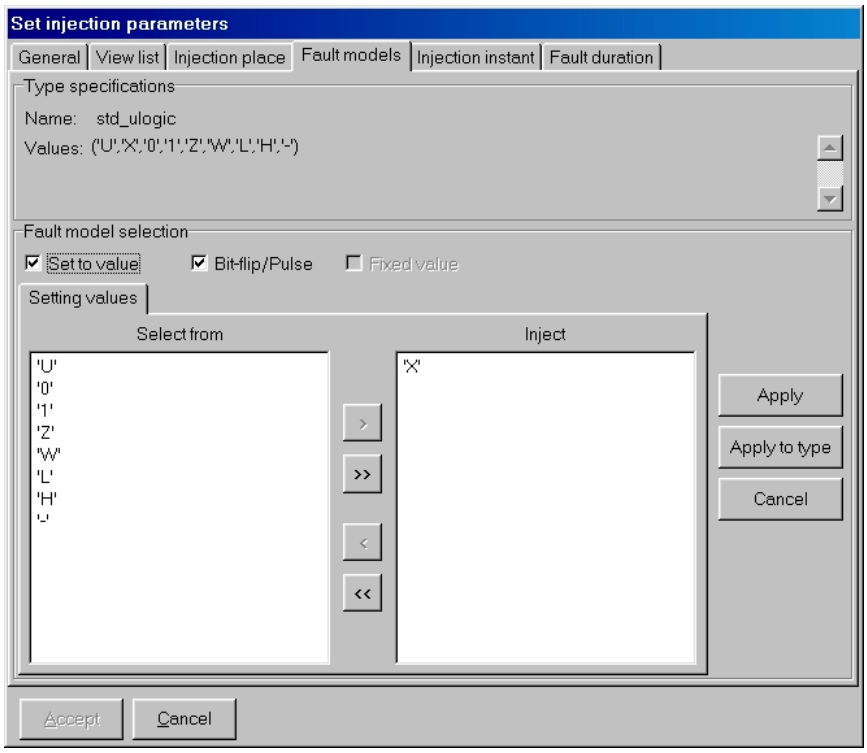

Figura 6.10: Especificación de los modelos de fallos *pulse* e *indetermination*. Paso 2: configuración de los modelos de fallos.

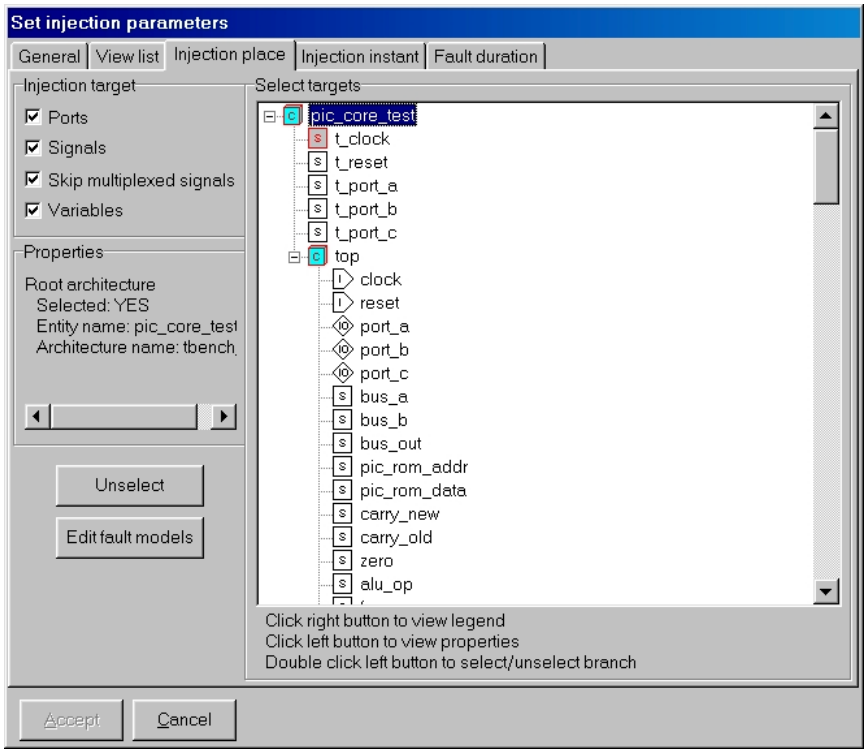

Figura 6.11: Especificación del modelo de fallo *delay*. Paso 1: elección del lugar de inyección.

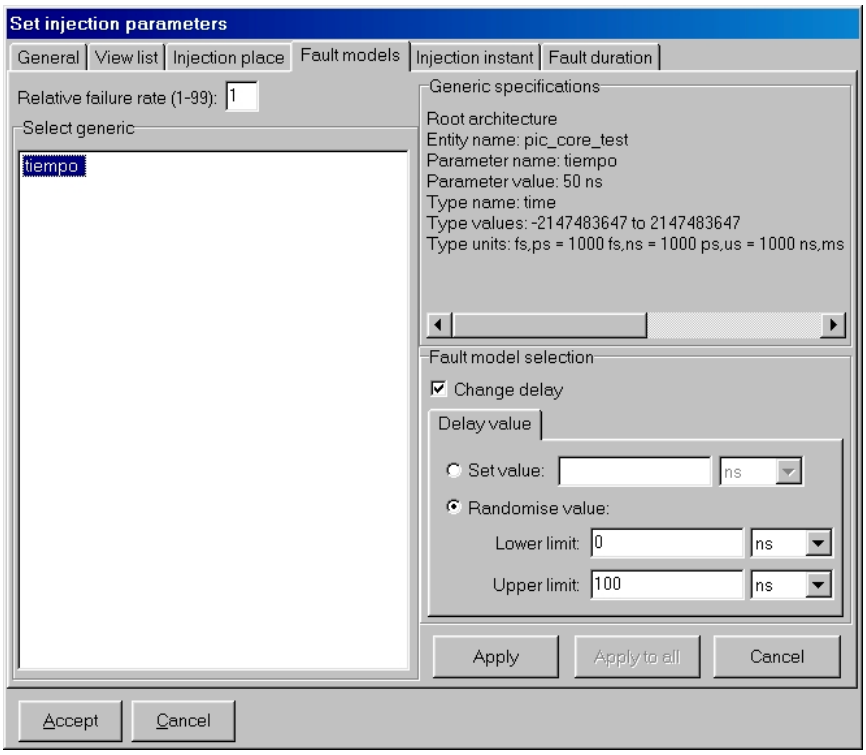

Figura 6.12: Especificación del modelo de fallo *delay*. Paso 2: configuración del modelo de fallos.

6.5.1.2 Parámetros del análisis

Como ya se ha indicado, el objetivo del experimento de inyección es el estudio del síndrome de error del procesador. Sin embargo, no se trata de un estudio convencional, en el que se desea estudiar el efecto de cada tipo de fallo ni los tipos de errores producidos. El experimento que aquí se presenta se enclava en el estudio de la representatividad de los modelos de fallos a nivel lógico y RT. Por este motivo, en los parámetros del análisis sólo interesa ver cómo se propagan los fallos inyectados a nivel lógico (en los circuitos combinacionales) en el nivel RT (en los registros del procesador). Para ello hay que configurar los siguientes elementos:

1. **Tipo de análisis**: Estudio del Síndrome de error.
2. **Clasificación de fallos**: Se distinguen dos clases de fallos: DELAY (que incluye los fallos inyectados de tipo *Delay* –en la señal de reloj del sistema) y ALU (que engloba a todos los fallos inyectados en la ALU, de tipos *Pulse* e *Indetermination*).
3. **Clasificación de errores**: No importa, pero como es un parámetro obligatorio se establece un único grupo, llamado Error.
4. **Cláusulas de clasificación de errores**: No importan, ya que en este experimento se considera un error cualquier discrepancia entre la simulación sin y con fallos. Como es un parámetro obligatorio se cumplimenta con una ecuación cualquiera.

Las ventanas mostradas desde la Figura 6.13 hasta la Figura 6.15 indican la configuración correspondiente (para los parámetros especificados de 1 a 3).

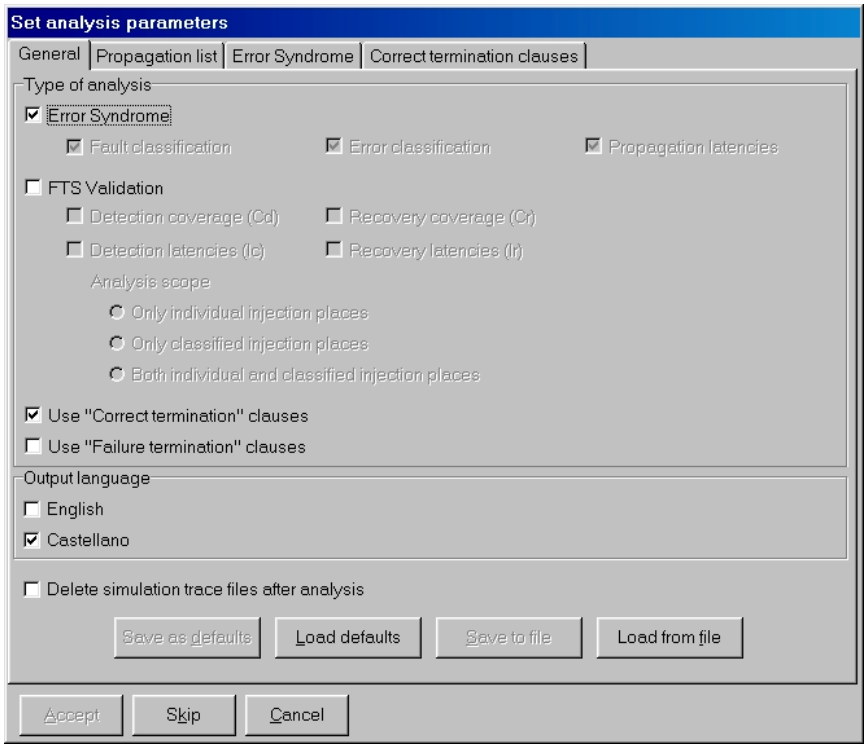

Figura 6.13: Características generales del análisis.

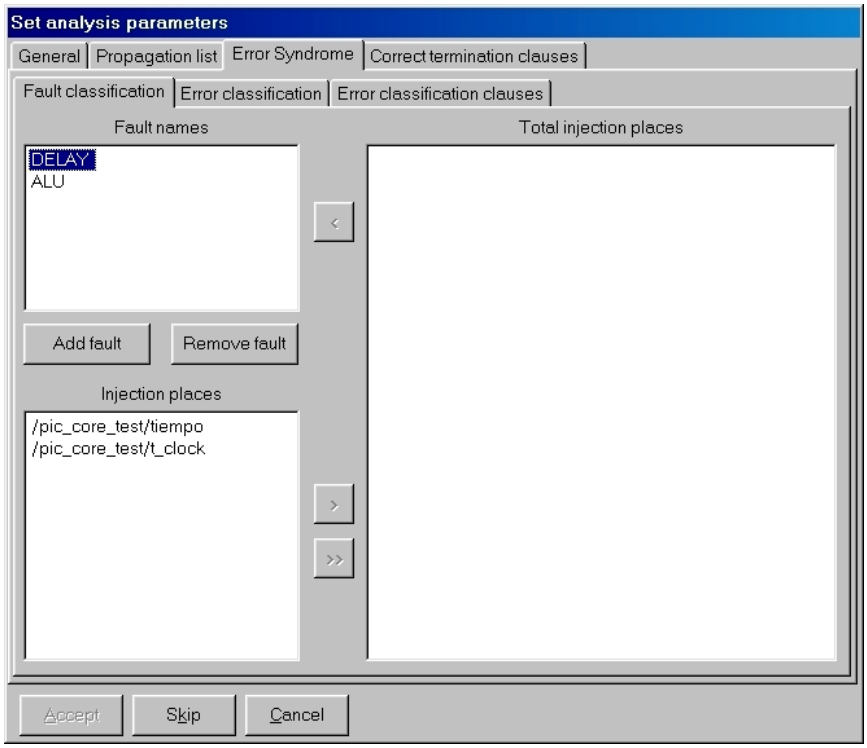

Figura 6.14: Clasificación de los fallos.

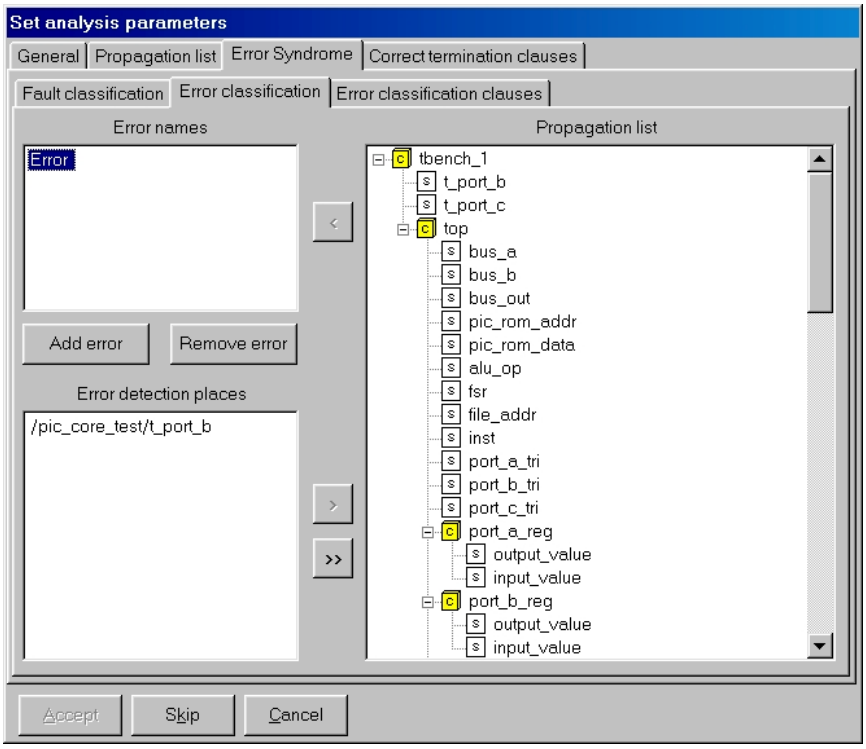

Figura 6.15: Clasificación de los errores.

6.5.1.3 Macro de inyección generada

A continuación se muestra un fragmento de la *macro* de inyección generada para realizar el experimento:

```
# Injection no. 1
do {D:\\Modelos VHDL\\PICStructural\\__Commands\\Dur_0-1T_T_Bubblesort_10\\InitModel.do}
run @1226
do {C:\\VFIT\\system\\Inject.do} 0 /pic_core_test/top/alu/adder__4/condicion2/adder_n/ s3 0 X 80
run @82000
write list D:\\MODELO~1\\PICSTR~1\\__COMM~1\\DUR_0-~2\\Fault00000.lst

# Injection no. 2
do {D:\\Modelos VHDL\\PICStructural\\__Commands\\Dur_0-1T_T_Bubblesort_10\\InitModel.do}
run @2811
do {C:\\VFIT\\system\\Inject.do} 0 /pic_core_test/top/alu/comp2/ i5 0 X 92
run @82000
write list D:\\MODELO~1\\PICSTR~1\\__COMM~1\\DUR_0-~2\\Fault00001.lst

# Injection no. 3
do {D:\\Modelos VHDL\\PICStructural\\__Commands\\Dur_0-1T_T_Bubblesort_10\\InitModel.do}
run @13746
do {C:\\VFIT\\system\\Inject.do} 0 /pic_core_test/top/alu/comp0/ i3 1 0 1 51
run @82000
write list D:\\MODELO~1\\PICSTR~1\\__COMM~1\\DUR_0-~2\\Fault00002.lst

# Injection no. 4
do {D:\\Modelos VHDL\\PICStructural\\__Commands\\Dur_0-1T_T_Bubblesort_10\\InitModel.do}
run @48479
do {C:\\VFIT\\system\\Inject.do} 0 /pic_core_test/top/alu/operador_bitset/decodificador/s2/ aux 1 0 1 11
run @82000
write list D:\\MODELO~1\\PICSTR~1\\__COMM~1\\DUR_0-~2\\Fault00003.lst

# Injection no. 5
do {D:\\Modelos VHDL\\PICStructural\\__Commands\\Dur_0-1T_T_Bubblesort_10\\InitModel.do}
run @34528
do {C:\\VFIT\\system\\Inject.do} 0 /pic_core_test/top/alu/ b_aux(5) 0 X 17
run @82000
write list D:\\MODELO~1\\PICSTR~1\\__COMM~1\\DUR_0-~2\\Fault00004.lst

...
```

```
# Injection no. 2996
do {D:\\Modelos VHDL\\PICStructural\\__Commands\\Dur_0-1T_T_Bubblesort_10\\InitModel.do}
run @20252
do {C:\\VFIT\\system\\Inject.do} 0 /pic_core_test/noname_process_0/ tiempo 3 19 ns 58
run @82000
write list D:\\MODELO~1\\PICSTR~1\\__COMM~1\\DUR_0-~2\\Fault02995.lst

# Injection no. 2997
do {D:\\Modelos VHDL\\PICStructural\\__Commands\\Dur_0-1T_T_Bubblesort_10\\InitModel.do}
run @8383
do {C:\\VFIT\\system\\Inject.do} 0 /pic_core_test/top/alu/adder__7/condicion2/adder_n/ s1 0 X 90
run @82000
write list D:\\MODELO~1\\PICSTR~1\\__COMM~1\\DUR_0-~2\\Fault02996.lst

# Injection no. 2998
do {D:\\Modelos VHDL\\PICStructural\\__Commands\\Dur_0-1T_T_Bubblesort_10\\InitModel.do}
run @23863
do {C:\\VFIT\\system\\Inject.do} 0 /pic_core_test/top/alu/operador_bitclr/decodificador/s0/ aux 1 0 1 55
run @82000
write list D:\\MODELO~1\\PICSTR~1\\__COMM~1\\DUR_0-~2\\Fault02997.lst

# Injection no. 2999
do {D:\\Modelos VHDL\\PICStructural\\__Commands\\Dur_0-1T_T_Bubblesort_10\\InitModel.do}
run @58273
do {C:\\VFIT\\system\\Inject.do} 0 /pic_core_test/top/alu/adder__3/condicion2/adder_n/ s2 1 0 1 48
run @82000
write list D:\\MODELO~1\\PICSTR~1\\__COMM~1\\DUR_0-~2\\Fault02998.lst

# Injection no. 3000
do {D:\\Modelos VHDL\\PICStructural\\__Commands\\Dur_0-1T_T_Bubblesort_10\\InitModel.do}
run @69866
do {C:\\VFIT\\system\\Inject.do} 0 /pic_core_test/top/alu/adder__2/condicion2/adder_n/ s1 0 X 61
run @82000
write list D:\\MODELO~1\\PICSTR~1\\__COMM~1\\DUR_0-~2\\Fault02999.lst
```

En la *macro* se pueden observar diferentes invocaciones a las *macros* InitModel.do e Inject.do. La primera sirve para indicar al simulador en cada simulación las señales que se deben incluir en la traza. La segunda es la *macro* general de inyección, perteneciente a la librería de *macros* (ver apartado 6.3.4).

6.5.1.4 Ficheros de resultados

A continuación se muestra un fragmento del fichero de resultados que VFIT genera. Dada su gran extensión, se ha reducido notablemente.

```
   # Descripción del experimento: Technique: Commands -- Fault duration: U[0.1T-1.0T] -- Workload: Bubble-
sort (n=10)
   # Fecha del análisis: 1/8/2003 Hora del análisis:  0:47:42

   # Análisis de las inyecciones por objetivos individuales

   # Inyecciones en Señales


              Clasificación de errores
   Nombre       N° fallos inyectados     N° fallos propagados     % fallos propagados      Latencia media
        N° regs. afectados       Multiplicidad    N° averías fallos no propagados   %  averías  fallos  no
propagados       N° averías fallos propagados      % averías fallos propagados       N° averías       %
averías  {Clase de error  N° errores propagados    Latencia media   N° regs. afectados       Multiplicidad}
        {Clase de error  N° errores propagados    Latencia media   N° regs. afectados       Multiplicidad}

   /pic_core_test/t_clock        89       60       67.42    39.97    564      9.40     27       30.34    1
        1.12     28       31.46    <<No Clasificados/Non Classified>>        556      42.76    556      1.00
        Error1   8        19.38    8        1.00
   /pic_core_test/top/alu/result 25       15       60.00    28.00    41       2.73     0        0.00     6
        24.00    6        24.00    <<No Clasificados/Non Classified>>        41       25.32    41       1.00
        --       --       --       --       --
   /pic_core_test/top/alu/temp_carry      35       2        5.71     36.50    3        1.50     0        0.00
        1        2.86     1        2.86     <<No Clasificados/Non Classified>>        3        39.33    3
        1.00     --       --       --       --       --

   ...

   /pic_core_test/top/alu/comp2/i4        21       9        42.86    26.67    11       1.22     0        0.00
        1        4.76     1        4.76     <<No Clasificados/Non Classified>>        11       23.64    11
        1.00     --       --       --       --       --
```

```
   /pic_core_test/top/alu/comp2/i5        21      2       9.52    21.00   2       1.00    0       0.00
        0       0.00    0       0.00    <<No Clasificados/Non Classified>>      2       21.00   2
        1.00    --      --      --      --      --
   /pic_core_test/top/alu/comp2/i6        31      8       25.81   47.38   12      1.50    0       0.00
        1       3.23    1       3.23    <<No Clasificados/Non Classified>>      12      57.67   12
        1.00    --      --      --      --      --


   # Inyecciones en Puertos y Señales


                Clasificación de errores
   Nombre       N° fallos inyectados     N° fallos propagados     % fallos propagados      Latencia media
        N° regs. afectados      Multiplicidad   N° averías fallos no propagados  %  averías  fallos  no
propagados      N° averías fallos propagados     % averías fallos propagados      N° averías      %
averías {Clase de error  N° errores propagados    Latencia media   N° regs. afectados      Multiplicidad}
        {Clase de error  N° errores propagados    Latencia media   N° regs. afectados      Multiplicidad}

   /pic_core_test/t_clock         89      60      67.42   39.97   564     9.40    27      30.34   1
        1.12    28      31.46   <<No Clasificados/Non Classified>>      556     42.76   556     1.00
        Error1  8       19.38   8       1.00
   /pic_core_test/top/alu/result  25      15      60.00   28.00   41      2.73    0       0.00    6
        24.00   6       24.00   <<No Clasificados/Non Classified>>      41      25.32   41      1.00
        --      --      --      --      --
   /pic_core_test/top/alu/temp_carry      35      2       5.71    36.50   3       1.50    0       0.00
        1       2.86    1       2.86    <<No Clasificados/Non Classified>>      3       39.33   3
        1.00    --      --      --      --      --

   ...

   /pic_core_test/top/alu/comp2/i4        21      9       42.86   26.67   11      1.22    0       0.00
        1       4.76    1       4.76    <<No Clasificados/Non Classified>>      11      23.64   11
        1.00    --      --      --      --      --
   /pic_core_test/top/alu/comp2/i5        21      2       9.52    21.00   2       1.00    0       0.00
        0       0.00    0       0.00    <<No Clasificados/Non Classified>>      2       21.00   2
        1.00    --      --      --      --      --
   /pic_core_test/top/alu/comp2/i6        31      8       25.81   47.38   12      1.50    0       0.00
        1       3.23    1       3.23    <<No Clasificados/Non Classified>>      12      57.67   12
        1.00    --      --      --      --      --


   # Inyecciones en Constantes genéricas


                Clasificación de errores
   Nombre       N° fallos inyectados     N° fallos propagados     % fallos propagados      Latencia media
        N° regs. afectados      Multiplicidad   N° averías fallos no propagados  %  averías  fallos  no
propagados      N° averías fallos propagados     % averías fallos propagados      N° averías      %
averías {Clase de error  N° errores propagados    Latencia media   N° regs. afectados      Multiplicidad}
        {Clase de error  N° errores propagados    Latencia media   N° regs. afectados      Multiplicidad}

   /pic_core_test/tiempo 95       69      72.63   101.36  728     10.55   0       0.00    0       0.00
        0       0.00    <<No Clasificados/Non Classified>>      728     100.08  728     1.00    --
        --      --      --      --


   # Análisis por tipos de objetivos individuales

                Clasificación de errores
   Tipo de objetivo     N° fallos inyectados     N° fallos propagados     % fallos propagados
        Latencia media   N° regs. afectados      Multiplicidad   N° averías fallos no propagados  %
averías fallos no propagados     N° averías fallos propagados     % averías fallos propagados      N°
averías  % averías       {Clase de error  N° errores propagados    Latencia media   N° regs. afectados
        Multiplicidad}   {Clase de error  N° errores propagados    Latencia media   N° regs. afectados
        Multiplicidad}
   Puertos      0       --      --      --      --      --      --      --      --      --
        --      --      --      --      --      --      --      --      --      --      --
   Señales      2905    250     8.61    30.39   1031    4.12    27      0.93    51      1.76    78
        2.69    Error1  8       19.38   8       1.00    <<No Clasificados/Non Classified>>      1023
        34.98   1023    1.00
   Puertos y Señales    2905    250     8.61    30.39   1031    4.12    27      0.93    51      1.76
        78      2.69    Error1  8       19.38   8       1.00    <<No Clasificados/Non Classified>>
        1023    34.98   1023    1.00
   Variables    0       --      --      --      --      --      --      --      --      --      --
        --      --      --      --      --      --      --      --      --      --      --
   Constantes genéricas 95      69      72.63   101.36  728     10.55   0       0.00    0       0.00
        0       0.00    Error1  0       0.00    0       0.00    <<No Clasificados/Non Classified>>
        728     100.08  728     1.00

   # Análisis por módulos

   # Inyecciones en componentes
```

```
                Clasificación de errores
   Nombre       N° fallos inyectados     N° fallos propagados     % fallos propagados      Latencia media
        N° regs. afectados       Multiplicidad    N° averías fallos no propagados   %  averías   fallos   no
propagados       N° averías fallos propagados      % averías fallos propagados       N° averías       %
averías {Clase de error  N° errores propagados    Latencia media   N° regs. afectados       Multiplicidad}
        {Clase de error  N° errores propagados    Latencia media   N° regs. afectados       Multiplicidad}

   /pic_core_test        184      129     70.11    72.81    1292     10.02    27       14.67    1        0.54
        28       15.22    <<No Clasificados/Non Classified>>        1284     75.26    1284     1.00     Error1
        8        19.38    8        1.00
   /pic_core_test/top/alu        570      56       9.82     27.12    130      2.32     0        0.00     17
        2.98     17       2.98     <<No Clasificados/Non Classified>>        130      25.81    130      1.00
        --       --       --       --       --
   /pic_core_test/top/alu/adder__7/condicion2/adder_n      80       9        11.25    32.00    25       2.78
        0        0.00     2        2.50     2        2.50     <<No Clasificados/Non Classified>>        25
        35.36    25       1.00     --       --       --       --       --

   ...

   /pic_core_test/top/alu/comp0  242      7        2.89     34.43    28       4.00     0        0.00     5
        2.07     5        2.07     <<No Clasificados/Non Classified>>        28       34.43    28       1.00
        --       --       --       --       --
   /pic_core_test/top/alu/comp1  228      0        0.00     0.00     0        0.00     0        0.00     0
        0.00     0        0.00     --       --       --       --       --       --       --       --
        --
   /pic_core_test/top/alu/comp2  226      33       14.60    34.61    45       1.36     0        0.00     4
        1.77     4        1.77     <<No Clasificados/Non Classified>>        45       36.58    45       1.00
        --       --       --       --       --


   # Análisis por tipos de módulos


                Clasificación de errores
   Tipo de módulo        N° fallos inyectados     N° fallos propagados     % fallos propagados
        Latencia media   N° regs. afectados       Multiplicidad    N° averías fallos no propagados   %
averías fallos no propagados      N° averías fallos propagados      % averías fallos propagados       N°
averías  % averías       {Clase de error  N° errores propagados    Latencia media   N° regs. afectados
        Multiplicidad}   {Clase de error  N° errores propagados    Latencia media   N° regs. afectados
        Multiplicidad}
   Componentes   3000     319      10.63    45.74    1759     5.51     27       0.90     51       1.70     78
        2.60     Error1   8        19.38    8        1.00     <<No Clasificados/Non Classified>>        1751
        62.05    1751     1.00
   Procesos      0        --       --       --       --       --       --       --       --       --       --
        --       --       --       --       --       --       --       --       --       --
   Bloques       0        --       --       --       --       --       --       --       --       --       --
        --       --       --       --       --       --       --       --       --       --
   Sentencias GENERATE   0        --       --       --       --       --       --       --       --       --
        --       --       --       --       --       --       --       --       --       --       --


   # Análisis por clases de fallos

        Clasificación de errores
   Clase de fallo        N° fallos inyectados     N° fallos propagados     % fallos propagados
        Latencia Media   N° regs. afectados       Multiplicidad    N° averías fallos no propagados   %
averías fallos no propagados      N° averías fallos propagados      % averías fallos propagados       N°
averías  % averías       {Clase de error  N° errores propagados    Latencia media   N° regs. afectados
        Multiplicidad}   {Clase de error  N° errores propagados    Latencia media   N° regs. afectados
        Multiplicidad}
   ALU  2816     190      6.75     27.37    467      2.46     0        0.00     50       1.78     50       1.78
        <<No Clasificados/Non Classified>>        467      25.72    467      1.00     --       --       --
        --       --
   DELAY184      129      70.11    72.81    1292     10.02    27       14.67    1        0.54     28       15.22
        Error1   8        19.38    8        1.00     <<No Clasificados/Non Classified>>        1284     75.26
        1284     1.00


   # Análisis general


                Clasificación de errores
   N° fallos inyectados  N° fallos propagados     % fallos propagados      Latencia media   N° regs. afec-
tados    Multiplicidad    N° averías fallos no propagados   % averías fallos no propagados    N°      averías
fallos propagados         % averías fallos propagados       N° averías       % averías       {Clase de error
        N° errores propagados    Latencia media   N° regs. afectados       Multiplicidad}   {Clase de error
        N° errores propagados    Latencia media   N° regs. afectados       Multiplicidad}
   3000 319      10.63    0.14     1759     5.51     27       0.90     51       1.70     78       2.60     Error1
        8        19.38    8        1.00     <<No Clasificados/Non Classified>>        1751     62.05    1751
        1.00
```

```
   # Intervalos de confianza de errores activados
   % fallos propagados   Intervalo de confianza al 95%     Intervalo de confianza al 99%
   10.630.03     0.04

   # Latencias
   MediaMínima   Máxima   Mediana  Varianza de las latencias Intervalo de confianza al 95%     Intervalo    de
confianza al 99%
   45.741        199      34       1600.75  4.39     5.77
```

Como se puede comprobar, la información se genera en forma de tablas, para ser interpretada de manera sencilla por otras utilidades, como hojas de cálculo (Excel, GNU plot, etc.) o procesadores de texto. En estas tablas se tratan básicamente los mismos datos, pero desde diferentes puntos de vista. Los datos calculados se refieren al estudio de la evolución de los fallos desde el momento de la inyección, y son, entre otros:

- Los números y porcentajes de fallos propagados.
- Sus latencias de propagación medias.
- La cantidad de registros afectados y la multiplicidad (o número promedio de registros alterados por cada fallo propagado; véase el apartado 7.5.1.4 para una mayor información).
- Las averías ocasionadas, tanto por los fallos propagados como por los no propagados.

Además, estos datos se indican tanto de forma general como teniendo en cuenta la clasificación de errores realizada en la fase de configuración (es decir, desglosándolos para cada clase de error).

Por otro lado, la existencia de múltiples tablas se debe a que cada una muestra los datos indicados desde distintos puntos de vista, o aspectos. Se exponen clasificándolos en función de:

- Los lugares de inyección, tanto de forma individual (cada señal, puerto, variable, etc.) como resumida (los datos de todas las señales, puertos, variables, etc.).
- Por los tipos de módulos, o bloques lógicos, individual (cada componente, bloque, proceso, etc.) y globalmente (todos los componentes, bloques, procesos, etc.).
- Los tipos de fallo, atendiendo a la clasificación de fallos realizada en la fase de configuración.
- De forma global, sin distinguir ninguna clasificación.

Los datos expuestos en las últimas tablas reflejan la validez estadística de los experimentos, pues muestran los valores medios de los fallos propagados y de la latencia de propagación con un intervalo de confianza. En el ejemplo mostrado arriba aparecen dos intervalos de confianza, uno calculado para un grado de confianza del 95 % y otro para un gradode confianza del 98 %).

6.5.2 Ejemplo de validación de un sistema tolerante a fallos

Como ya se ha explicado, en este apartado se muestran los pasos que hay que seguir para llevar a cabo un experimento de inyección orientado a la validación de un sistema tolerante a fallos.

El experimento se realiza sobre el modelo en VHDL de un sistema computador tolerante a fallos basado en el procesador MARK2 (descrito en el apartado 7.2.2). En los próximos apartados se describen los parámetros del experimento (distinguiendo entre los relativos a la inyección y los relativos al análisis), la macro de inyección generada y los ficheros de resultados obtenidos.

6.5.2.1 Parámetros de inyección

Los parámetros de inyección son:

1. **Número de fallos inyectados**: 3000.
2. **Duración del ciclo de reloj (T)**: 1000 ns.
3. **Carga de trabajo**: Suma de la serie aritmética de *n* números enteros, con $n = 6$.
4. **Duración de la carga de trabajo (T_W)**: 264 μs.
5. **Duración de la simulación (T_S)**: 600 μs.
6. **Lugares de inyección**: Se pueden inyectar fallos en todas las señales y variables del modelo excepto las relacionadas con la CPU de repuesto (CPUB).
7. **Instante de inyección**: Se genera de manera aleatoria, siguiendo una distribución Uniforme entre 0.0 ns y T_W (T_W = 264 μs).
8. **Duración de los fallos**: Se inyectan fallos transitorios, cuya duración sigue una distribución Uniforme entre 0.1 T y 10.0 T, donde T es la duración del ciclo de reloj del modelo, especificado en 2.
9. **Modelos de fallos**: Los fallos que se inyectan son *stuck-at*[70] ('0', '1'), *bit-flip*, *indetermination* y *delay*.

El modo en que estos parámetros se introducen en VFIT se muestra en las ventanas que aparecen desde la Figura 6.16 a la Figura 6.23.

[70] En la época en que se realizaron originalmente estos experimentos, aún no se había definido el modelo *pulse* para distinguirlo del *bit-flip* (véase el capítulo 4). En su lugar, se empleaban los modelos *stuck-at* transitorios.

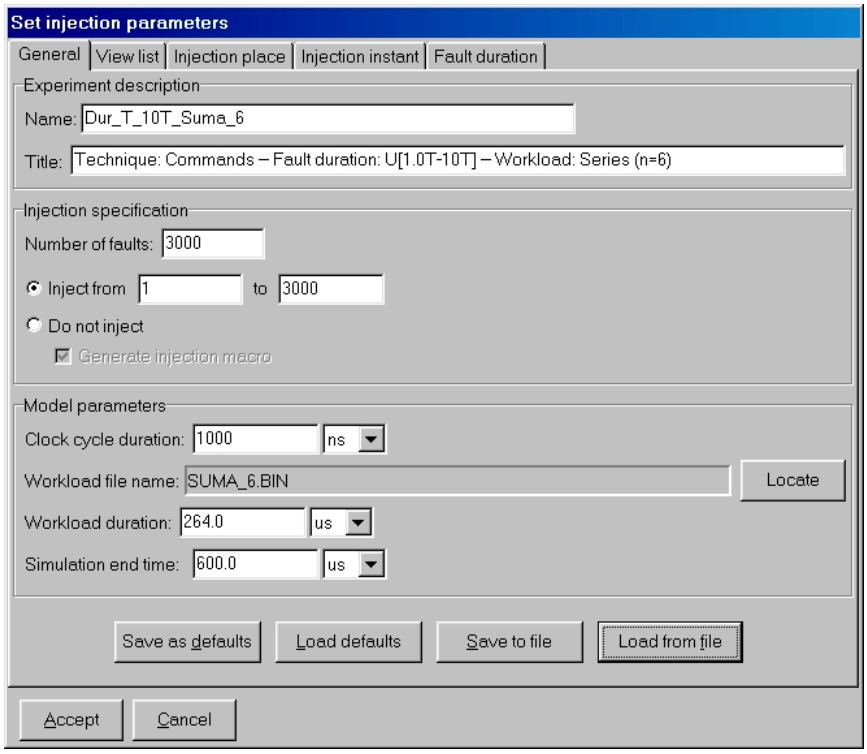

Figura 6.16: Especificación del número de fallos inyectados, la duración del ciclo de reloj, la carga de trabajo y las duraciones de la carga y de la simulación completa.

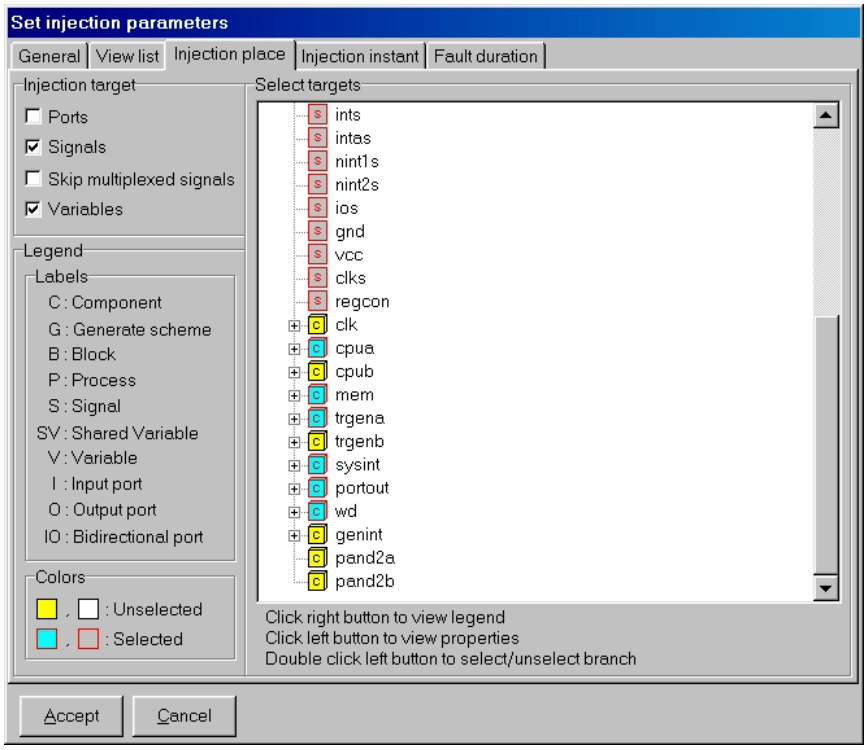

Figura 6.17: Selección de los lugares de inyección.

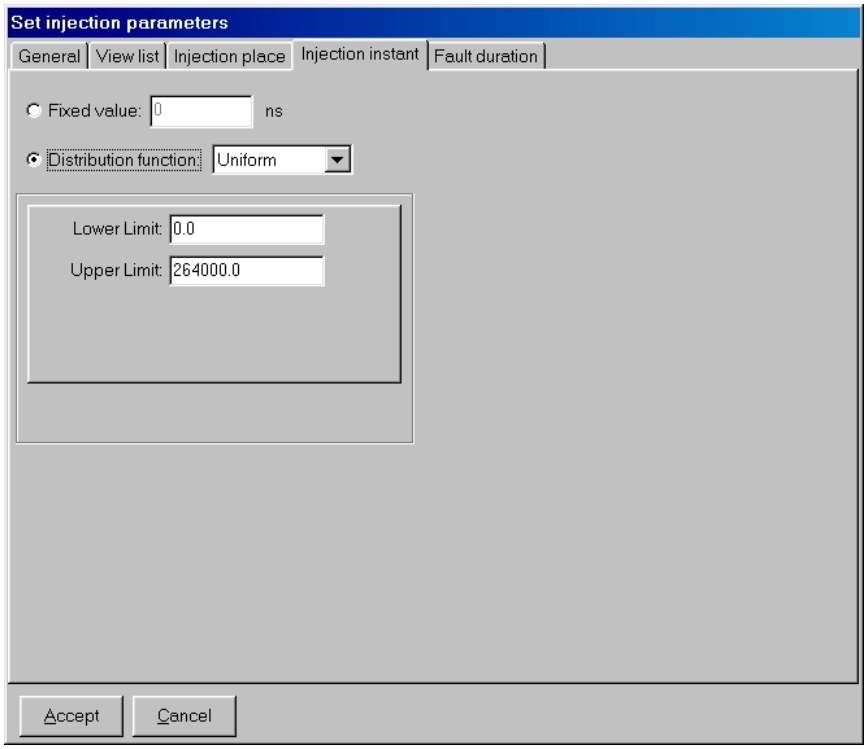

Figura 6.18: Especificación del instante de inyección.

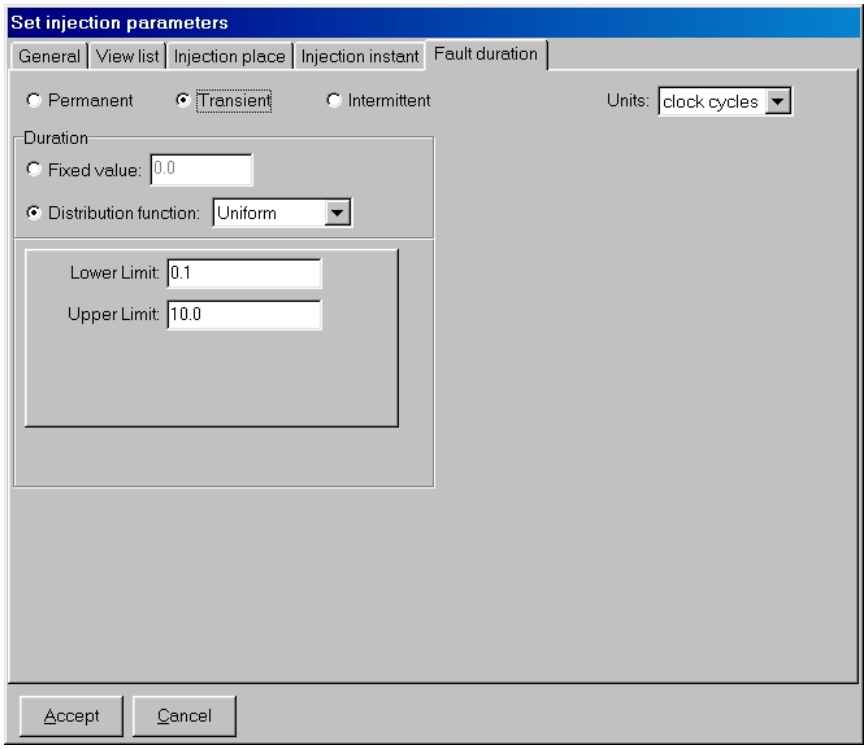

Figura 6.19: Especificación de la duración de los fallos.

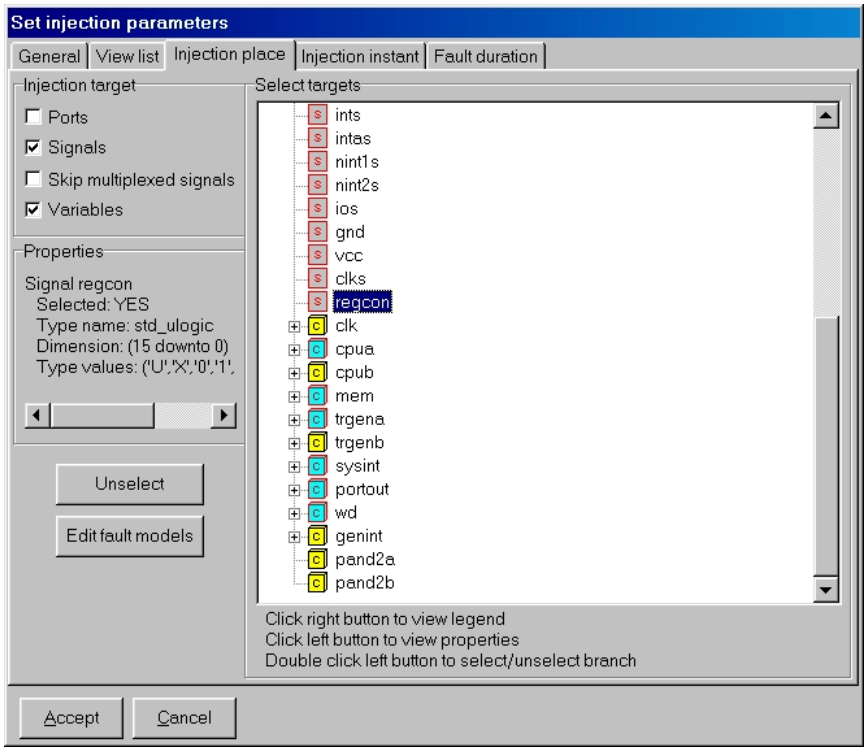

Figura 6.20: Especificación de los modelos de fallos *stuck-at*, *bit-flip* e *indetermination*. Paso 1: elección del objetivo de inyección.

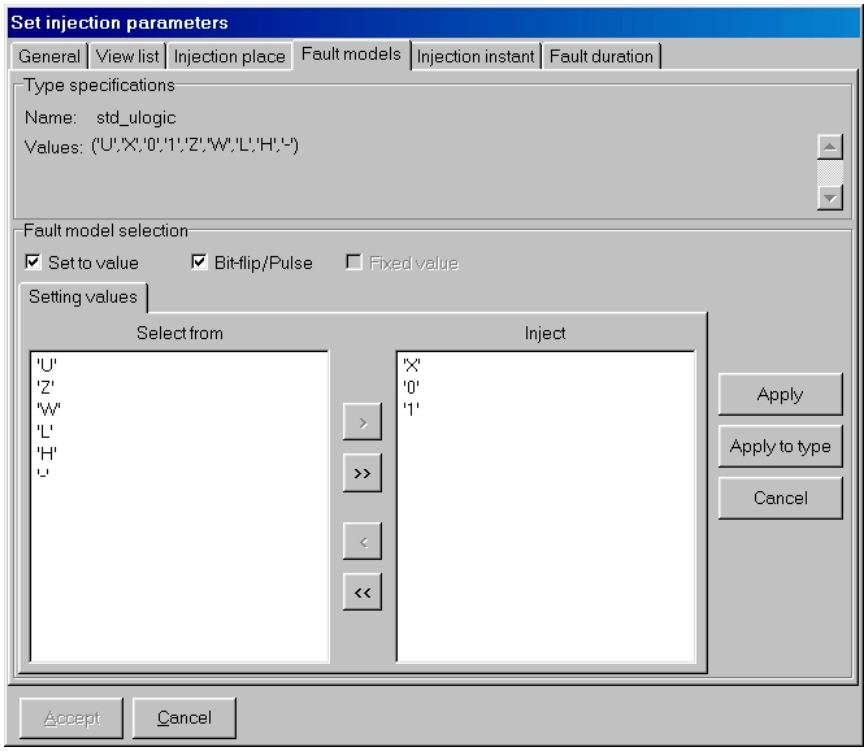

Figura 6.21: Especificación de los modelos de fallos *stuck-at*, *bit-flip* e *indetermination*. Paso 2: configuración de los modelos de fallos.

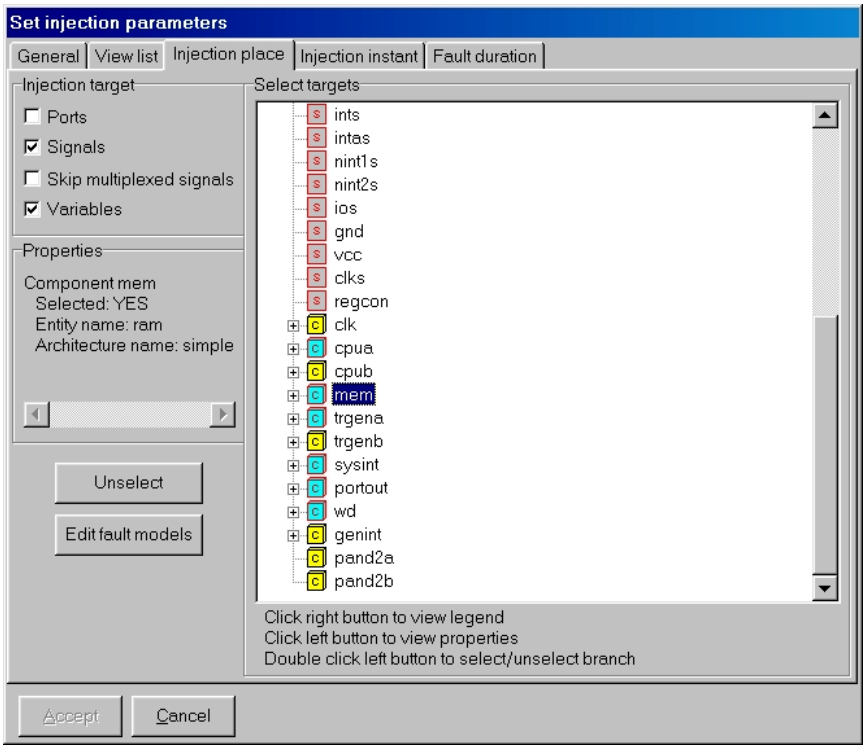

Figura 6.22: Especificación del modelo de fallo *delay*. Paso 1: elección del lugar de inyección.

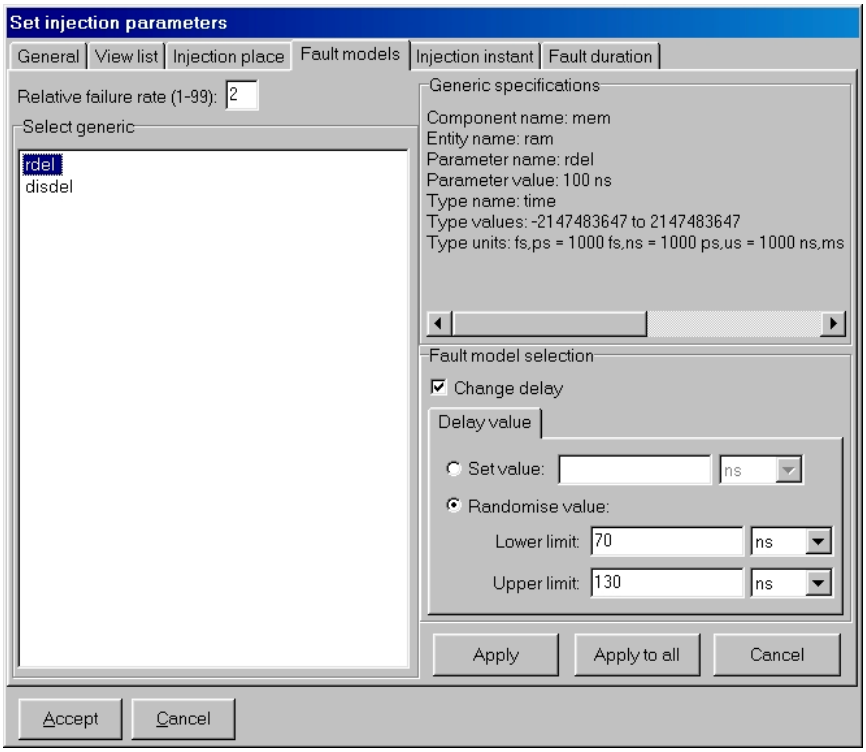

Figura 6.23: Especificación del modelo de fallo *delay*. Paso 2: configuración del modelo de fallos.

En cuanto al análisis, éste es completamente diferente al realizado en el ejemplo anterior. En este caso, se lleva a cabo la validación de los mecanismos de tolerancia a fallos. Para ello hay que configurar los siguientes elementos:

1. **Tipo de análisis**: Validación de un STF.
2. **Relación de mecanismos de tolerancia a fallos**:
   - Mecanismos de detección: Dos mecanismos de paridad (uno en cada CPU, llamados ParityA y ParityB) y un temporizador de guardia (Watchdog).
   - Mecanismos de recuperación: Por un lado, en cada procesador se implementan sendos ciclos de recuperación[71] (llamados BackoffA y BackoffB). Por otro, asociado al temporizador de guardia hay un mecanismo de puntos de comprobación (o, *Checkpointing* llamado Checkpoint). Por último, hay un mecanismo que reemplaza la CPU principal por la de repuesto (llamado Spare).
3. **Cláusulas de detección de errores**: Determinan las condiciones bajo las cuales un determinado mecanismo de detección detecta la existencia de un error.
4. **Cláusulas de recuperación de errores**: Determinan las condiciones bajo las cuales un determinado mecanismo de recuperación recupera un error detectado.

Las ventanas mostradas desde la Figura 6.24 hasta la Figura 6.27 indican la configuración correspondiente a los parámetros 1 a 3.

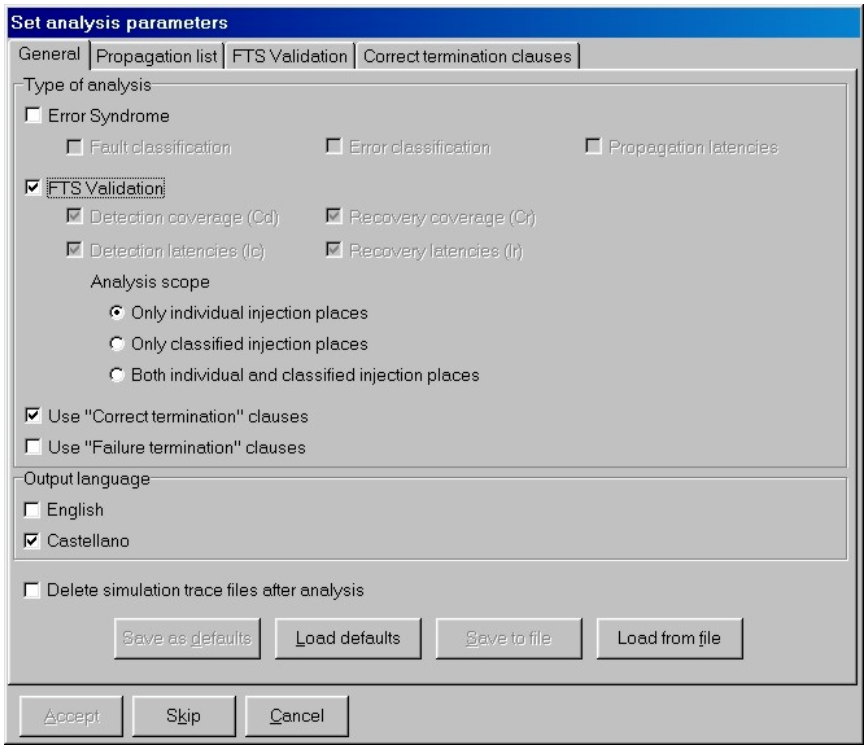

Figura 6.24: Características generales del análisis.

[71] O ciclos de *Backoff* (véase el apartado 7.2.2).

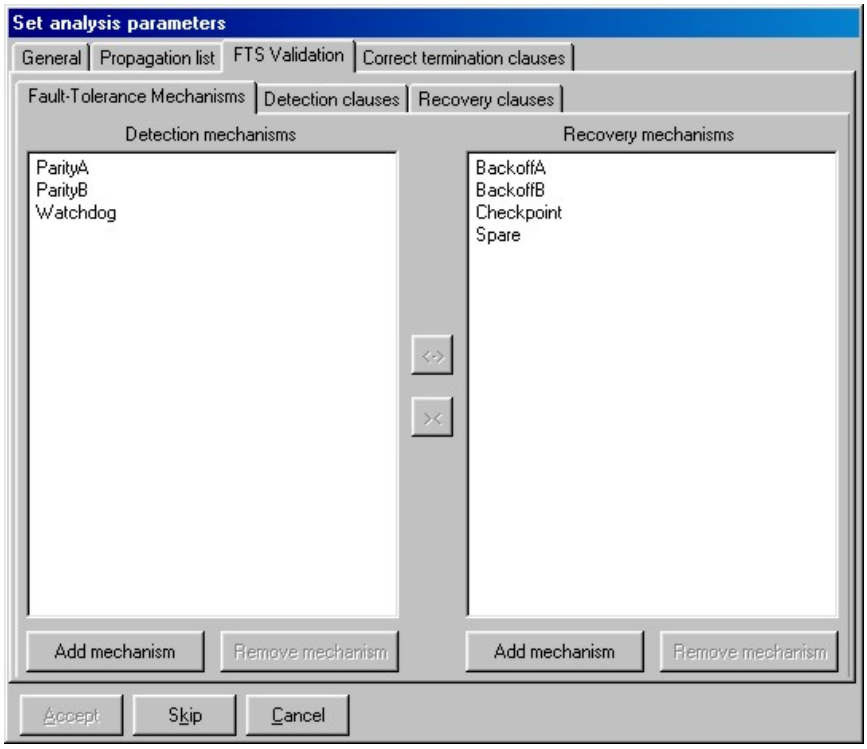

Figura 6.25: Especificación de los mecanismos de tolerancia a fallos.

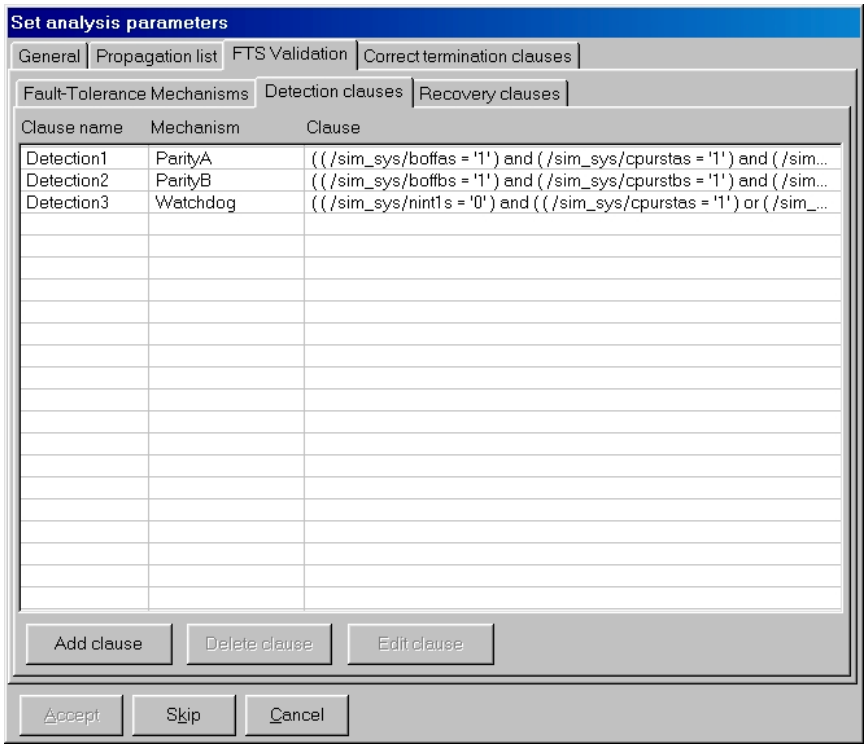

Figura 6.26: Cláusulas de detección de errores.

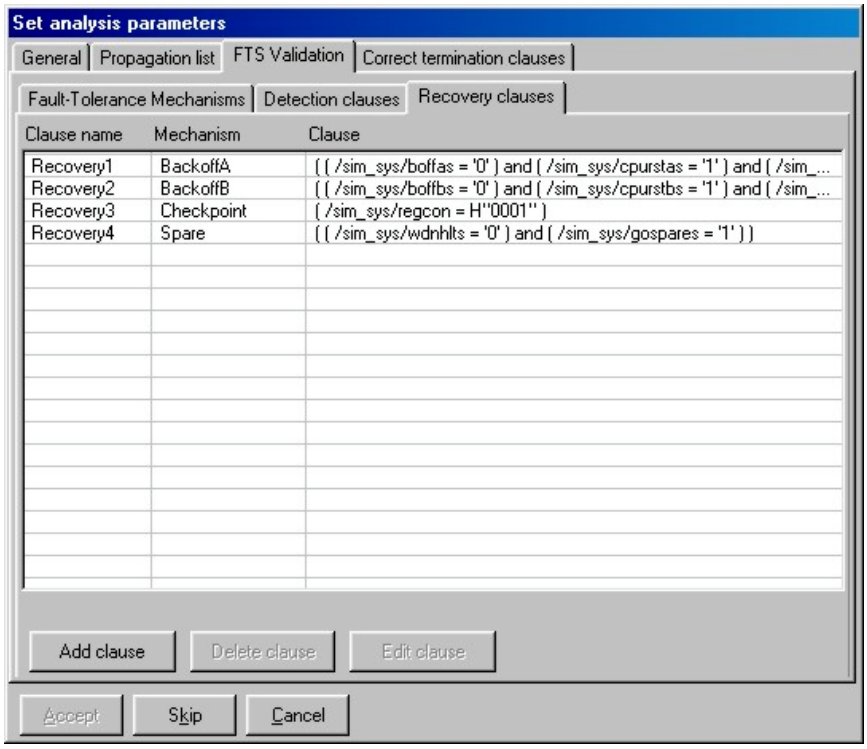

Figura 6.27: Cláusulas de recuperación de errores.

6.5.2.2 Macro de inyección generada

A continuación se muestra un fragmento de la *macro* de inyección generada para realizar el experimento:

```
# Injection no. 1
do {D:\\Modelos VHDL\\Markstdl\\__Commands\\Dur_T_10T_Suma_6\\InitModel.do}
run @4048
do {C:\\VFIT\\system\\Inject.do} 0 /sim_sys/cpua/noname_process_0/ wdel 3 112 ns 7778
run @600000
write list D:\\MODELO~1\\Markstdl\\__COMM~1\\DUR_T_~1\\Fault00000.lst

# Injection no. 2
do {D:\\Modelos VHDL\\Markstdl\\__Commands\\Dur_T_10T_Suma_6\\InitModel.do}
run @9279
do {C:\\VFIT\\system\\Inject.do} 1 /sim_sys/wd/timer_pro/ count 2 2 9101
run @600000
write list D:\\MODELO~1\\Markstdl\\__COMM~1\\DUR_T_~1\\Fault00001.lst

# Injection no. 3
do {D:\\Modelos VHDL\\Markstdl\\__Commands\\Dur_T_10T_Suma_6\\InitModel.do}
run @120360
do {C:\\VFIT\\system\\Inject.do} 1 /sim_sys/mem/memory/ mem(72)(14) 0 0 8145
run @600000
write list D:\\MODELO~1\\Markstdl\\__COMM~1\\DUR_T_~1\\Fault00002.lst

# Injection no. 4
do {D:\\Modelos VHDL\\Markstdl\\__Commands\\Dur_T_10T_Suma_6\\InitModel.do}
run @22412
do {C:\\VFIT\\system\\Inject.do} 0 /sim_sys/ gnd 0 0 1452
run @600000
write list D:\\MODELO~1\\Markstdl\\__COMM~1\\DUR_T_~1\\Fault00003.lst

# Injection no. 5
do {D:\\Modelos VHDL\\Markstdl\\__Commands\\Dur_T_10T_Suma_6\\InitModel.do}
run @152992
do {C:\\VFIT\\system\\Inject.do} 0 /sim_sys/mem/noname_process_0/ disdel 3 24 ns 6003
run @600000
write list D:\\MODELO~1\\Markstdl\\__COMM~1\\DUR_T_~1\\Fault00004.lst

...
```

```
# Injection no. 2996
do {D:\\Modelos VHDL\\Markstdl\\__Commands\\Dur_T_10T_Suma_6\\InitModel.do}
run @208356
do {C:\\VFIT\\system\\Inject.do} 0 /sim_sys/ intas 0 X 4417
run @600000
write list D:\\MODELO~1\\Markstdl\\__COMM~1\\DUR_T_~1\\Fault02995.lst

# Injection no. 2997
do {D:\\Modelos VHDL\\Markstdl\\__Commands\\Dur_T_10T_Suma_6\\InitModel.do}
run @19502
do {C:\\VFIT\\system\\Inject.do} 0 /sim_sys/portout/noname_process_0/ ffdel 3 38 ns 3410
run @600000
write list D:\\MODELO~1\\Markstdl\\__COMM~1\\DUR_T_~1\\Fault02996.lst

# Injection no. 2998
do {D:\\Modelos VHDL\\Markstdl\\__Commands\\Dur_T_10T_Suma_6\\InitModel.do}
run @178844
do {C:\\VFIT\\system\\Inject.do} 0 /sim_sys/cpua/noname_process_0/ odel 3 47 ns 6199
run @600000
write list D:\\MODELO~1\\Markstdl\\__COMM~1\\DUR_T_~1\\Fault02997.lst

# Injection no. 2999
do {D:\\Modelos VHDL\\Markstdl\\__Commands\\Dur_T_10T_Suma_6\\InitModel.do}
run @5624
do {C:\\VFIT\\system\\Inject.do} 0 /sim_sys/sysint/noname_process_0/ bufdel 3 19 ns 4123
run @600000
write list D:\\MODELO~1\\Markstdl\\__COMM~1\\DUR_T_~1\\Fault02998.lst

# Injection no. 3000
do {D:\\Modelos VHDL\\Markstdl\\__Commands\\Dur_T_10T_Suma_6\\InitModel.do}
run @244271
do {C:\\VFIT\\system\\Inject.do} 0 /sim_sys/ pchkbs 0 X 5904
run @600000
write list D:\\MODELO~1\\Markstdl\\__COMM~1\\DUR_T_~1\\Fault02999.lst
```

6.5.2.3 Ficheros de resultados

El formato de los resultados generados por VFIT se puede ver en el fragmento de fichero que se muestra a continuación.

```
   # Descripción del experimento: Technique: Commands -- Fault duration: U[1.0T-10.0T] -- Workload: Series
(n=6)
   # Fecha del análisis: 1/8/2003 Hora del análisis: 11:57:04

   # Análisis de las inyecciones por objetivos individuales

   # Inyecciones en Señales
   Nombre       N° fallos inyectados     N° fallos no activados   % fallos no activados    N° fallos acti-
vados    % fallos activados       Latencia media propagación       N° errores no efectivos  %  errores  no
efectivos       N° errores no detectados (averia) % errores no detectados  N° errores detectados
       Cobertura detección mecanismos    Cobertura detección sistema       Latencia media detección
       {Mecanismo de detección  N° errores detectados     % errores detectados     Cobertura detección
       Latencia media detección} {Mecanismo de detección  N° errores detectados     % errores detectados
       Cobertura detección       Latencia media detección} {Mecanismo de detección  N° errores detectados
       % errores detectados      Cobertura detección       Latencia media detección} N° errores no recupera-
dos (averia)     % errores no recuperados (averia) N° errores no recuperados (sin averia)    %  errores  no
recuperados (sin averia) N° errores recuperados   % errores recuperados    Cobertura recuperación mecanis-
mos      Cobertura recuperación sistema    Latencia media recuperación       {Mecanismo de recuperación
       N° errores recuperados   % errores recuperados    Cobertura recuperación   Latencia  media  recupe-
ración} {Mecanismo de recuperación       N° errores recuperados   % errores recuperados    Cobertura recu-
peración Latencia media recuperación}     {Mecanismo de recuperación       N° errores recuperados   %
errores recuperados      Cobertura recuperación   Latencia media recuperación}      {Mecanismo de recupera-
ción    N° errores recuperados   % errores recuperados    Cobertura recuperación   Latencia  media  recupe-
ración}
   /sim_sys/data_bus     35       4        11.43    31       88.57    122.16   3        9.68     0        0.00
        28       90.32    100.00   820.36   ParityA  28       4.86     90.32    820.36   --       --       --
        --       --       --       --       --       --       --       3        10.71    0        0.00     25
        89.29    80.65    90.32    103504.80         BackoffA 11       2.51     35.48    15863.64 Spare    14
        3.19     45.16    172365.71         --       --       --       --       --       --       --       --
        --       --
   /sim_sys/bit_par      40       3        7.50     37       92.50    94.81    4        10.81    0        0.00
        33       89.19    100.00   592.12   ParityA  33       5.73     89.19    592.12   --       --       --
        --       --       --       --       --       --       --       0        0.00     0        0.00     33
        100.00   89.19    100.00   45918.79 BackoffA 25       5.69     67.57    4987.20  Spare    8        1.82
        21.62    173830.00         --       --       --       --       --       --       --       --       --
        --
   /sim_sys/add_bus      49       2        4.08     47       95.92    292.40   14       29.79    4        8.51
        29       61.70    91.49    29190.00 Watchdog 6        1.04     12.77    124756.67         ParityA  23
        3.99     48.94    4259.57  --       --       --       --       --       4        13.79    0        0.00
        25       86.21    53.19    82.98    90403.20 Checkpoint       4        0.91     8.51     110.00
```

```
        BackoffA 9        2.05     19.15    26333.33 Spare    12       2.73     25.53    168553.33        --
        --       --       --       --
    ...
    /sim_sys/portout/srq  29       29       100.00   0        0.00     0.00     0        0.00     0        0.00
        0        0.00     0.00     0.00     --       --       --       --       --       --       --       --
        --       --       --       --       --       --       --       0        0.00     0        0.00     0
        0.00     0.00     0.00     0.00     --       --       --       --       --       --       --       --
        --       --       --       --       --       --       --       --       --       --       --       --

    /sim_sys/portout/q    53       14       26.42    39       73.58    124.72   37       94.87    0        0.00
        2        5.13     100.00   73135.00 Watchdog 2        0.35     5.13     73135.00 --       --       --
        --       --       --       --       --       --       --       0        0.00     2        100.00   0
        0.00     0.00     94.87    0.00     --       --       --       --       --       --       --       --
        --       --       --       --       --       --       --       --       --       --       --       --

    /sim_sys/portout/b1/guard     47       12       25.53    35       74.47    464.06   23       65.71    0
        0.00     12       34.29    100.00   102701.66         Watchdog 12       2.08     34.29    102701.67
        --       --       --       --       --       --       --       --       --       0        0.00
        7        58.33    5        41.67    14.29    80.00    40416.00 Checkpoint       3        0.68     8.57
        20.00    Spare    2        0.46     5.71     101010.00         --       --       --       --       --
        --       --       --       --       --


    # Inyecciones en Puertos y Señales
    Nombre        N° fallos inyectados     N° fallos no activados   % fallos no activados    N° fallos acti-
vados    % fallos activados        Latencia media propagación        N° errores no efectivos  %  errores  no
efectivos        N° errores no detectados (averia) % errores no detectados  N° errores detectados
        Cobertura detección mecanismos    Cobertura detección sistema       Latencia media detección
        {Mecanismo de detección  N° errores detectados     % errores detectados     Cobertura detección
        Latencia media detección} {Mecanismo de detección  N° errores detectados    % errores detectados
        Cobertura detección      Latencia media detección} {Mecanismo de detección  N° errores detectados
        % errores detectados     Cobertura detección      Latencia media detección} N° errores no recupera-
dos (averia)     % errores no recuperados (averia) N° errores no recuperados (sin averia)    %  errores  no
recuperados (sin averia) N° errores recuperados   % errores recuperados    Cobertura recuperación mecanis-
mos      Cobertura recuperación sistema    Latencia media recuperación       {Mecanismo de recuperación
        N° errores recuperados   % errores recuperados    Cobertura recuperación   Latencia  media  recupe-
ración} {Mecanismo de recuperación        N° errores recuperados   % errores recuperados    Cobertura recu-
peración Latencia media recuperación}     {Mecanismo de recuperación       N° errores recuperados   %
errores recuperados      Cobertura recuperación   Latencia media recuperación}     {Mecanismo de recupera-
ción    N° errores recuperados   % errores recuperados    Cobertura recuperación   Latencia  media  recupe-
ración}
    /sim_sys/data_bus     35       4        11.43    31       88.57    122.16   3        9.68     0        0.00
        28       90.32    100.00   820.36   ParityA  28       4.86     90.32    820.36   --       --       --
        --       --       --       --       --       --       --       3        10.71    0        0.00     25
        89.29    80.65    90.32    103504.80         BackoffA 11       2.51     35.48    15863.64 Spare    14
        3.19     45.16    172365.71         --       --       --       --       --       --       --       --
        --       --
    /sim_sys/bit_par      40       3        7.50     37       92.50    94.81    4        10.81    0        0.00
        33       89.19    100.00   592.12   ParityA  33       5.73     89.19    592.12   --       --       --
        --       --       --       --       --       --       --       0        0.00     0        0.00     33
        100.00   89.19    100.00   45918.79 BackoffA 25       5.69     67.57    4987.20  Spare    8        1.82
        21.62    173830.00         --       --       --       --       --       --       --       --       --
        --
    /sim_sys/add_bus      49       2        4.08     47       95.92    292.40   14       29.79    4        8.51
        29       61.70    91.49    29190.00 Watchdog 6        1.04     12.77    124756.67         ParityA  23
        3.99     48.94    4259.57  --       --       --       --       --       4        13.79    0        0.00
        25       86.21    53.19    82.98    90403.20 Checkpoint       4        0.91     8.51     110.00
        BackoffA 9        2.05     19.15    26333.33 Spare    12       2.73     25.53    168553.33        --
        --       --       --       --
    ...
    /sim_sys/portout/srq  29       29       100.00   0        0.00     0.00     0        0.00     0        0.00
        0        0.00     0.00     0.00     --       --       --       --       --       --       --       --
        --       --       --       --       --       --       --       0        0.00     0        0.00     0
        0.00     0.00     0.00     0.00     --       --       --       --       --       --       --       --
        --       --       --       --       --       --       --       --       --       --       --       --

    /sim_sys/portout/q    53       14       26.42    39       73.58    124.72   37       94.87    0        0.00
        2        5.13     100.00   73135.00 Watchdog 2        0.35     5.13     73135.00 --       --       --
        --       --       --       --       --       --       --       0        0.00     2        100.00   0
        0.00     0.00     94.87    0.00     --       --       --       --       --       --       --       --
        --       --       --       --       --       --       --       --       --       --       --       --

    /sim_sys/portout/b1/guard     47       12       25.53    35       74.47    464.06   23       65.71    0
        0.00     12       34.29    100.00   102701.66         Watchdog 12       2.08     34.29    102701.67
        --       --       --       --       --       --       --       --       --       0        0.00
        7        58.33    5        41.67    14.29    80.00    40416.00 Checkpoint       3        0.68     8.57
        20.00    Spare    2        0.46     5.71     101010.00         --       --       --       --       --
        --       --       --       --       --


    # Inyecciones en Variables
    Nombre        N° fallos inyectados     N° fallos no activados   % fallos no activados    N° fallos acti-
vados    % fallos activados        Latencia media propagación        N° errores no efectivos  %  errores  no
efectivos        N° errores no detectados (averia) % errores no detectados  N° errores detectados
```

```
        Cobertura detección mecanismos    Cobertura detección sistema       Latencia media detección
        {Mecanismo de detección  N° errores detectados     % errores detectados      Cobertura detección
        Latencia media detección} {Mecanismo de detección  N° errores detectados     % errores detectados
        Cobertura detección       Latencia media detección} {Mecanismo de detección  N° errores detectados
        % errores detectados      Cobertura detección       Latencia media detección} N° errores no recupera-
dos (averia)     % errores no recuperados (averia) N° errores no recuperados (sin averia)     %  errores  no
recuperados (sin averia) N° errores recuperados    % errores recuperados     Cobertura recuperación mecanis-
mos      Cobertura recuperación sistema    Latencia media recuperación       {Mecanismo de recuperación
        N° errores recuperados    % errores recuperados     Cobertura recuperación    Latencia  media  recupe-
ración}  {Mecanismo de recuperación        N° errores recuperados    % errores recuperados     Cobertura recu-
peración Latencia media recuperación}      {Mecanismo de recuperación        N° errores recuperados    %
errores recuperados       Cobertura recuperación    Latencia media recuperación}      {Mecanismo de recupera-
ción     N° errores recuperados    % errores recuperados     Cobertura recuperación    Latencia  media  recupe-
ración}
    /sim_sys/mem/memory/mem        89       89       100.00   0        0.00     0.00     0        0.00     0
         0.00     0        0.00     0.00     0.00     --       --       --       --       --       --       --
         --       --       --       --       --       --       --       --       0        0.00     0        0.00
         0        0.00     0.00     0.00     0.00     --       --       --       --       --       --       --
         --       --       --       --       --       --       --       --       --       --       --       --
         --
    /sim_sys/mem/memory/par        79       78       98.73    1        1.27     9658.00  0        0.00     0
         0.00     1        100.00   100.00   30.00    ParityA  1        0.17     100.00   30.00    --       --
         --       --       --       --       --       --       --       --       1        100.00   0        0.00
         0        0.00     0.00     0.00     0.00     --       --       --       --       --       --       --
         --       --       --       --       --       --       --       --       --       --       --       --
         --
    /sim_sys/wd/timer_pro/count    37       37       100.00   0        0.00     0.00     0        0.00     0
         0.00     0        0.00     0.00     0.00     --       --       --       --       --       --       --
         --       --       --       --       --       --       --       --       0        0.00     0        0.00
         0        0.00     0.00     0.00     0.00     --       --       --       --       --       --       --
         --       --       --       --       --       --       --       --       --       --       --       --
         --
    /sim_sys/wd/timer_pro/hlt      40       40       100.00   0        0.00     0.00     0        0.00     0
         0.00     0        0.00     0.00     0.00     --       --       --       --       --       --       --
         --       --       --       --       --       --       --       --       0        0.00     0        0.00
         0        0.00     0.00     0.00     0.00     --       --       --       --       --       --       --
         --       --       --       --       --       --       --       --       --       --       --       --
         --


    # Inyecciones en Genéricos
    Nombre       N° fallos inyectados     N° fallos no activados   % fallos no activados    N° fallos acti-
vados    % fallos activados       Latencia media propagación       N° errores no efectivos  %  errores  no
efectivos       N° errores no detectados (averia) % errores no detectados  N° errores detectados
        Cobertura detección mecanismos    Cobertura detección sistema       Latencia media detección
        {Mecanismo de detección  N° errores detectados     % errores detectados      Cobertura detección
        Latencia media detección} {Mecanismo de detección  N° errores detectados     % errores detectados
        Cobertura detección       Latencia media detección} {Mecanismo de detección  N° errores detectados
        % errores detectados      Cobertura detección       Latencia media detección} N° errores no recupera-
dos (averia)     % errores no recuperados (averia) N° errores no recuperados (sin averia)     %  errores  no
recuperados (sin averia) N° errores recuperados    % errores recuperados     Cobertura recuperación mecanis-
mos      Cobertura recuperación sistema    Latencia media recuperación       {Mecanismo de recuperación
        N° errores recuperados    % errores recuperados     Cobertura recuperación    Latencia  media  recupe-
ración}  {Mecanismo de recuperación        N° errores recuperados    % errores recuperados     Cobertura recu-
peración Latencia media recuperación}      {Mecanismo de recuperación        N° errores recuperados    %
errores recuperados       Cobertura recuperación    Latencia media recuperación}      {Mecanismo de recupera-
ción     N° errores recuperados    % errores recuperados     Cobertura recuperación    Latencia  media  recupe-
ración}
    /sim_sys/cpua/rdel     40       0        0.00     40       100.00   649.85   31       77.50    0        0.00
         9        22.50    100.00   350.00   ParityA  9        1.56     22.50    350.00   --       --       --
         --       --       --       --       --       --       --       1        11.11    0        0.00     8
         88.89    20.00    97.50    90198.50 BackoffA 4        0.91     10.00    4637.00  Spare    4        0.91
         10.00    175760.00         --       --       --       --       --       --       --       --       --
         --
    /sim_sys/cpua/wdel     54       10       18.52    44       81.48    2494.52  44       100.00   0        0.00
         0        0.00     100.00   0.00     --       --       --       --       --       --       --       --
         --       --       --       --       --       --       --       0        0.00     0        0.00     0
         0.00     0.00     0.00     0.00     --       --       --       --       --       --       --       --
         --       --       --       --       --       --       --       --       --       --       --       --

    /sim_sys/cpua/odel     43       0        0.00     43       100.00   323.44   39       90.70    3        6.98
         1        2.33     93.02    7140.00  ParityA  1        0.17     2.33     7140.00  --       --       --
         --       --       --       --       --       --       --       0        0.00     0        0.00     1
         100.00   2.33     93.02    168830.00         Spare    1        0.23     2.33     168830.00         --
         --       --       --       --       --       --       --       --       --       --       --       --
         --       --
    ...
    /sim_sys/portout/gdel  35       35       100.00   0        0.00     0.00     0        0.00     0        0.00
         0        0.00     0.00     0.00     --       --       --       --       --       --       --       --
         --       --       --       --       --       --       --       0        0.00     0        0.00     0
         0.00     0.00     0.00     0.00     --       --       --       --       --       --       --       --
         --       --       --       --       --       --       --       --       --       --       --       --
```

```
   /sim_sys/portout/ffdel          32      26      81.25   6       18.75   4206.67 6       100.00  0
        0.00    0       0.00    100.00  0.00    --      --      --      --      --      --      --
        --      --      --      --      --      --      --      --      0       0.00    0       0.00
        0       0.00    0.00    0.00    0.00    --      --      --      --      --      --      --
        --      --      --      --      --      --      --      --      --      --      --      --
        --
   /sim_sys/portout/bufdel         46      41      89.13   5       10.87   1448.60 5       100.00  0
        0.00    0       0.00    100.00  0.00    --      --      --      --      --      --      --
        --      --      --      --      --      --      --      --      0       0.00    0       0.00
        0       0.00    0.00    0.00    0.00    --      --      --      --      --      --      --
        --      --      --      --      --      --      --      --      --      --      --      --
        --


   # Análisis por tipos de objetivos individuales

   Tipo de objetivo     N° fallos inyectados    N° fallos no activados  % fallos no activados   N° fallos activados   % fallos activados      Latencia media propagación      N° errores no efectivos  % errores no efectivos     N° errores no detectados (averia) % errores no detectados  N° errores detectados
        Cobertura detección mecanismos  Cobertura detección sistema     Latencia media detección
        {Mecanismo de detección  N° errores detectados   % errores detectados    Cobertura detección
        Latencia media detección} {Mecanismo de detección  N° errores detectados   % errores detectados
        Cobertura detección     Latencia media detección} {Mecanismo de detección  N° errores detectados
        % errores detectados    Cobertura detección     Latencia media detección} N° errores no recuperados (averia)    % errores no recuperados (averia) N° errores no recuperados (sin averia)    % errores no recuperados (sin averia) N° errores recuperados   % errores recuperados    Cobertura recuperación mecanismos     Cobertura recuperación sistema   Latencia media recuperación     {Mecanismo de recuperación
        N° errores recuperados   % errores recuperados    Cobertura recuperación  Latencia media recuperación}    {Mecanismo de recuperación      N° errores recuperados   % errores recuperados    Cobertura recuperación Latencia media recuperación}     {Mecanismo de recuperación      N° errores recuperados  % errores recuperados    Cobertura recuperación  Latencia media recuperación}    {Mecanismo de recuperación    N° errores recuperados   % errores recuperados    Cobertura recuperación  Latencia media recuperación}
   Puertos      0       0       0.00    0       0.00    0.00    0       0.00    0       0.00    0
        0.00    0.00    0.0     --      --      --      --      --      --      --      --
        --      --      --      --      --      --      0       0.00    0       0.00    0       0.00
        0.0     0.0     0.0     --      --      --      --      --      --      --      --
        --      --      --      --      --      --
   Señales      1997    666     33.35   1331    66.65   950.75  742     55.75   28      2.10    561
        42.15   97.90   %2196.83 ParityA 334    59.54   25.09   3846.54 ParityB 0       0.00    0.00
        0.00    Watchdog 227    40.46   17.05   77273.07 108    19.25   26      4.63    427     76.11
        32.08   87.83   473.26  BackoffA 135    31.62   10.14   12915.39 BackoffB 0     0.00    0.00
        0.00    Checkpoint      44      10.30   3.31    31857.86 Spare  248     58.08   18.63
        148478.13
   Puertos y Señales    1997    666     33.35   1331    66.65   950.75  742     55.75   28      2.10
        561     42.15   97.90   %2196.83 ParityA 334    59.54   25.09   3846.54 ParityB 0       0.00
        0.00    0.00    Watchdog 227    40.46   17.05   77273.07 108    19.25   26      4.63    427
        76.11   32.08   87.83   473.26  BackoffA 135    31.62   10.14   12915.39 BackoffB 0     0.00
        0.00    0.00    Checkpoint      44      10.30   3.31    31857.86 Spare  248     58.08   18.63
        148478.13
   Variables    245     244     99.59   1       0.41    9658.00 0       0.00    0       0.00    1
        100.00  100.00  %30.00  ParityA 1       100.00  100.00  30.00   ParityB 0       0.00    0.00
        0.00    Watchdog 0      0.00    0.00    0.00    1       100.00  0       0.00    0       0.00
        0.00    0.00    0.0     --      --      --      --      --      --      --      --
        --      --      --      --      --      --
   Genéricos    758     391     51.58   367     48.42   1002.49 349     95.10   4       1.09    14
        3.81    98.91   %5465.71 ParityA 14     100.00  3.81    6200.71 ParityB 0       0.00    0.00
        0.00    Watchdog 0      0.00    0.00    0.00    2       14.29   0       0.00    12      85.71
        3.27    98.37   46624.17 BackoffA 4     33.33   1.09    4637.00 BackoffB 0      0.00    0.00
        0.00    Checkpoint      0       0.00    0.00    0.00    Spare   8       66.67   2.18
        178920.00


   # Análisis por módulos

   # Inyecciones en Componentes
   Nombre       N° fallos inyectados    N° fallos no activados  % fallos no activados   N° fallos activados   % fallos activados      Latencia media propagación      N° errores no efectivos  % errores no efectivos     N° errores no detectados (averia) % errores no detectados  N° errores detectados
        Cobertura detección mecanismos  Cobertura detección sistema     Latencia media detección
        {Mecanismo de detección  N° errores detectados   % errores detectados    Cobertura detección
        Latencia media detección} {Mecanismo de detección  N° errores detectados   % errores detectados
        Cobertura detección     Latencia media detección} {Mecanismo de detección  N° errores detectados
        % errores detectados    Cobertura detección     Latencia media detección} N° errores no recuperados (averia)    % errores no recuperados (averia) N° errores no recuperados (sin averia)    % errores no recuperados (sin averia) N° errores recuperados   % errores recuperados    Cobertura recuperación mecanismos     Cobertura recuperación sistema   Latencia media recuperación     {Mecanismo de recuperación
        N° errores recuperados   % errores recuperados    Cobertura recuperación  Latencia media recuperación}    {Mecanismo de recuperación      N° errores recuperados   % errores recuperados    Cobertura recuperación Latencia media recuperación}     {Mecanismo de recuperación      N° errores recuperados  % errores recuperados    Cobertura recuperación  Latencia media recuperación}    {Mecanismo de recuperación    N° errores recuperados   % errores recuperados    Cobertura recuperación  Latencia media recuperación}
```

```
   /sim_sys      1007     202      20.06    805      79.94    219.16   444      55.16    7        0.87     354
        43.98    99.13    23775.96 ParityA  243      42.19    30.19    4196.96  Watchdog 111      19.27    13.79
        66638.10 --       --       --       --       --       91       25.71    5        1.41     258      72.88
        32.05    87.20    76754.80 BackoffA 117      26.65    14.53    8851.09  Checkpoint        35       7.97
        4.35     30667.03 Spare    106      24.15    13.17    166922.77         --       --       --       --
        --
   /sim_sys/cpua 766      241      31.46    525      68.54    2284.27  316      60.19    25       4.76     184
        35.05    95.24    38206.38 Watchdog 81       14.06    15.43    82489.38 ParityA  103      17.88    19.62
        3381.88  --       --       --       --       --       19       10.33    2        1.09     163      88.59
        31.05    91.24    123499.27         Spare    137      31.21    26.10    139681.71         Checkpoint
        4        0.91     0.76     66860.00 BackoffA 22       5.01     4.19     33024.91 --       --       --
        --       --
   /sim_sys/mem  182      8        4.40     174      95.60    687.04   174      100.00   0        0.00     0
        0.00     100.00   0.00     --       --       --       --       --       --       --       --       --
        --       --       --       --       --       --       0        0.00     0        0.00     0        0.00
        0.00     0.00     0.00     --       --       --       --       --       --       --       --       --
        --       --       --       --       --       --       --       --       --       --       --
   /sim_sys/trgena        124      124      100.00   0        0.00     0.00     0        0.00     0        0.00
        0        0.00     0.00     0.00     --       --       --       --       --       --       --       --
        --       --       --       --       --       --       --       0        0.00     0        0.00     0
        0.00     0.00     0.00     0.00     --       --       --       --       --       --       --       --
        --       --       --       --       --       --       --       --       --       --       --       --

   /sim_sys/sysint        246      217      88.21    29       11.79    1967.34  27       93.10    0        0.00
        2        6.90     100.00   1680.00  ParityA  2        0.35     6.90     1680.00  --       --       --
        --       --       --       --       --       --       --       0        0.00     0        0.00     2
        100.00   6.90     100.00   156330.00         Spare    2        0.46     6.90     156330.00         --
        --       --       --       --       --       --       --       --       --       --       --       --
        --       --
   /sim_sys/portout       383      253      66.06    130      33.94    498.82   107      82.31    0        0.00
        23       17.69    100.00   96960.87 Watchdog 23       3.99     17.69    96960.87 --       --       --
        --       --       --       --       --       --       --       0        0.00     12       52.17    11
        47.83    8.46     90.77    88180.91 Spare    9        2.05     6.92     101010.00         Checkpoint
        2        0.46     1.54     30450.00 --       --       --       --       --       --       --       --
        --       --
```

```
   # Inyecciones en Procesos
   Nombre       N° fallos inyectados     N° fallos no activados   % fallos no activados    N° fallos acti-
vados    % fallos activados       Latencia media propagación       N° errores no efectivos  %  errores  no
efectivos       N° errores no detectados (averia) % errores no detectados  N° errores detectados
       Cobertura detección mecanismos    Cobertura detección sistema       Latencia media detección
       {Mecanismo de detección  N° errores detectados     % errores detectados     Cobertura detección
       Latencia media detección}{Mecanismo de detección  N° errores detectados     % errores detectados
       Cobertura detección       Latencia media detección}{Mecanismo de detección  N° errores detectados
       % errores detectados      Cobertura detección       Latencia media detección}N° errores no recupera-
dos (averia)     % errores no recuperados (averia) N° errores no recuperados (sin averia)    %  errores  no
recuperados (sin averia) N° errores recuperados   % errores recuperados    Cobertura recuperación mecanis-
mos      Cobertura recuperación sistema    Latencia media recuperación       {Mecanismo de recuperación
       N° errores recuperados   % errores recuperados    Cobertura recuperación   Latencia  media  recupe-
ración}  {Mecanismo de recuperación       N° errores recuperados   % errores recuperados    Cobertura recu-
peración Latencia media recuperación}     {Mecanismo de recuperación       N° errores recuperados   %
errores recuperados      Cobertura recuperación   Latencia media recuperación}      {Mecanismo de recupera-
ción    N° errores recuperados   % errores recuperados    Cobertura recuperación   Latencia  media  recupe-
ración}
   /sim_sys/mem/memory   168      167      99.40    1        0.60     9658.00  0        0.00     0        0.00
        1        100.00   100.00   30.00    ParityA  1        0.17     100.00   30.00    --       --       --
        --       --       --       --       --       --       --       1        100.00   0        0.00     0
        0.00     0.00     0.00     0.00     --       --       --       --       --       --       --       --
        --       --       --       --       --       --       --       --       --       --       --       --

   /sim_sys/wd/timer_pro 77       77       100.00   0        0.00     0.00     0        0.00     0        0.00
        0        0.00     0.00     0.00     --       --       --       --       --       --       --       --
        --       --       --       --       --       --       --       0        0.00     0        0.00     0
        0.00     0.00     0.00     0.00     --       --       --       --       --       --       --       --
        --       --       --       --       --       --       --       --       --       --       --       --
```

```
   # Inyecciones en Bloques
   Nombre       N° fallos inyectados     N° fallos no activados   % fallos no activados    N° fallos acti-
vados    % fallos activados       Latencia media propagación       N° errores no efectivos  %  errores  no
efectivos       N° errores no detectados (averia) % errores no detectados  N° errores detectados
       Cobertura detección mecanismos    Cobertura detección sistema       Latencia media detección
       {Mecanismo de detección  N° errores detectados     % errores detectados     Cobertura detección
       Latencia media detección}{Mecanismo de detección  N° errores detectados     % errores detectados
       Cobertura detección       Latencia media detección}{Mecanismo de detección  N° errores detectados
       % errores detectados      Cobertura detección       Latencia media detección}N° errores no recupera-
dos (averia)     % errores no recuperados (averia) N° errores no recuperados (sin averia)    %  errores  no
recuperados (sin averia) N° errores recuperados   % errores recuperados    Cobertura recuperación mecanis-
mos      Cobertura recuperación sistema    Latencia media recuperación       {Mecanismo de recuperación
       N° errores recuperados   % errores recuperados    Cobertura recuperación   Latencia  media  recupe-
ración}  {Mecanismo de recuperación       N° errores recuperados   % errores recuperados    Cobertura recu-
peración Latencia media recuperación}     {Mecanismo de recuperación       N° errores recuperados   %
```

```
errores recuperados      Cobertura recuperación   Latencia media recuperación}      {Mecanismo de recupera-
ción    N° errores recuperados   % errores recuperados    Cobertura recuperación   Latencia media recupe-
ración}
   /sim_sys/portout/b1   47      12      25.53   35      74.47   464.06  23      65.71   0       0.00
        12      34.29   100.00  102701.66       Watchdog 12     2.08    34.29   102701.67       --
        --      --      --      --      --      --      --      --      0       0.00    7
        58.33   5       41.67   14.29   80.00   40416.00 Checkpoint     3       0.68    8.57    20.00
        Spare   2       0.46    5.71    101010.00       --      --      --      --      --      --
        --      --      --      --


   # Análisis por tipos de módulos

   Clases de fallos     N° fallos inyectados     N° fallos no activados   % fallos no activados    N° fa-
llos activados   % fallos activados       Latencia media propagación       N° errores no efectivos  %
errores no efectivos     N° errores no detectados (averia) % errores no detectados  N° errores detectados
        Cobertura detección mecanismos    Cobertura detección sistema      Latencia media detección
        {Mecanismo de detección  N° errores detectados    % errores detectados     Cobertura detección
        Latencia media detección} {Mecanismo de detección  N° errores detectados    % errores detectados
        Cobertura detección      Latencia media detección} {Mecanismo de detección  N° errores detectados
        % errores detectados     Cobertura detección      Latencia media detección} N° errores no recupera-
dos (averia)     % errores no recuperados (averia) N° errores no recuperados (sin averia)    %  errores  no
recuperados (sin averia) N° errores recuperados   % errores recuperados    Cobertura recuperación mecanis-
mos      Cobertura recuperación sistema    Latencia media recuperación      {Mecanismo de recuperación
        N° errores recuperados   % errores recuperados    Cobertura recuperación   Latencia  media  recupe-
ración} {Mecanismo de recuperación       N° errores recuperados   % errores recuperados    Cobertura recu-
peración Latencia media recuperación}     {Mecanismo de recuperación       N° errores recuperados   %
errores recuperados      Cobertura recuperación   Latencia media recuperación}      {Mecanismo de recupera-
ción    N° errores recuperados   % errores recuperados    Cobertura recuperación   Latencia  media  recupe-
ración}
   Componentes  2708    1045    38.59   1663    61.41   972.41  1068    64.22   32      1.92    563
        33.85   98.08   3961.10 ParityA 348     61.81   20.93   3941.25 ParityB 0       0.00    0.00
        0.00    Watchdog 215    38.19   12.93   75853.81 110    19.54   19      3.37    434     77.09
        26.10   90.32   2235.00 BackoffA 139    32.03   8.36    12677.16 BackoffB 0     0.00    0.00
        0.00    Checkpoint       41      9.45    2.47    34187.46 Spare  254     58.53   15.27
        149810.74
   Procesos     245     244     99.59   1       0.41    9658.00 0       0.00    0       0.00    1
        100.00  100.00  30.00   ParityA 1       100.00  100.00  30.00   ParityB 0       0.00    0.00
        0.00    Watchdog 0      0.00    0.00    0.00    1       100.00  0       0.00    0       0.00
        0.00    0.00    0.0     --      --      --      --      --      --      --      --      --
        --      --      --      --      --      --
   Bloques      47      12      25.53   35      74.47   464.06  23      65.71   0       0.00    12
        34.29   100.00  102701.67       ParityA 0       0.00    0.00    0.00    ParityB 0       0.00
        0.00    0.00    Watchdog 12     100.00  34.29   102701.67       0       0.00    7       58.33
        5       41.67   14.29   80.00   40416.00 BackoffA 0     0.00    0.00    0.00    BackoffB 0
        0.00    0.00    0.00    Checkpoint       3       60.00   8.57    20.00   Spare   2       40.00
        5.71    101010.00
   Sentencias Generate  0       0       0.00    0       0.00    0.00    0       0.00    0       0.00
        0       0.00    0.00    0.0     --      --      --      --      --      --      --      --
        --      --      --      --      --      --      --      0       0.00    0       0.00    0
        0.00    0.0     0.0     0.0     --      --      --      --      --      --      --      --
        --      --      --      --      --      --      --

   # Análisis general
   N° fallos inyectados  N° fallos no activados   % fallos no activados    N° fallos activados      %  fa-
llos activados   Latencia media propagación       N° errores no efectivos  % errores no efectivos   N°
errores no detectados (averia)    % errores no detectados  N° errores detectados    Cobertura     detección
mecanismos       Cobertura detección sistema      Latencia media detección {Mecanismo de detección  N°
errores detectados       % errores detectados     Cobertura detección      Latencia media detección}
        {Mecanismo de detección  N° errores detectados    % errores detectados     Cobertura detección
        Latencia media detección} {Mecanismo de detección  N° errores detectados    % errores detectados
        Cobertura detección      Latencia media detección} N° errores no recuperados (averia)        %
errores no recuperados (averia)   N° errores no recuperados (sin averia)    % errores no recuperados (sin
averia) N° errores recuperados   % errores recuperados    Cobertura recuperación mecanismos Cobertura recu-
peración sistema Latencia media recuperación      {Mecanismo de recuperación       N° errores recuperados
        % errores recuperados    Cobertura recuperación   Latencia media recuperación}     {Mecanismo  de
recuperación     N° errores recuperados   % errores recuperados    Cobertura recuperación   Latencia  media
recuperación}    {Mecanismo de recuperación       N° errores recuperados   % errores recuperados
        Cobertura recuperación   Latencia media recuperación}     {Mecanismo de recuperación       N°
errores recuperados      % errores recuperados    Cobertura recuperación   Latencia media recuperación}

   3000 1301    43.37   1699    56.63   967.05  1091    64.21   32      1.88    576     33.90   98.12
        32834.33 ParityA 349    60.59   20.54   3930.04 ParityB 0       0.00    0.00    0.00
        Watchdog 227    39.41   13.36   77273.07 111    19.27   26      4.51    439     76.22   25.84
        90.05   94345.80 BackoffA 139    31.66   8.18    12677.16 BackoffB 0     0.00    0.00    0.00
        Checkpoint       44      10.02   2.59    31857.86 Spare  256     58.31   15.07   149429.47


   # Análisis de las latencias
   Tipo Media   Mínima  Máxima  Mediana Varianza Intervalo de confianza al 95%    Intervalo  de  confianza
al 99%
   Propagación  967.05  1       43425   195     10663538.00     155.28  204.08
   Detección    32834.33 9      420450  1660    2246449920.00   3870.73 5087.25
```

```
   Recuperación  94345.90 20      383320   101010   5715165184.00    7071.93  9294.54


   # Análisis de los mecanismos de detección
   Mecanismo     N° errores detectados     % errores detectados      Cobertura detección       Intervalo    de
confianza al 95% Intervalo de confianza al 99%     Latencia media   Latencia mínima  Latencia máxima
        Mediana de las latencias  Varianza de las latencias Intervalo de confianza al 95%     Intervalo    de
confianza al 99%
   ParityA       349      60.59    20.54    0.04     0.05     3930.04  22       70130    500      104283616.00
        1071.40  1408.12
   Watchdog      227      39.41    13.36    0.04     0.05     77273.08 9        420450   79000    2281254912.00
        6213.41  8166.20


   # Análisis de los mecanismos de recuperación
   Mecanismo     N° errores recuperados    % errores recuperados     Cobertura recuperación    Intervalo    de
confianza al 95% Intervalo de confianza al 99%     Latencia media   Latencia mínima  Latencia máxima
        Mediana de las latencias  Varianza de las latencias Intervalo de confianza al 95%     Intervalo    de
confianza al 99%
   BackoffA      139      31.66    8.18     0.04     0.06     12677.16 713      205000   5000     1063549760.00
        5421.60  7125.53
   Checkpoint    44       10.02    2.59     0.03     0.04     31857.86 20       267020   24085    1862453504.00
        12751.83 16759.55
   Spare256      58.31    15.07    0.05     0.06     .149429.597656   101010   383320   161080   1571444352.00
        4856.08  6382.27
```

Al igual que sucede en el estudio del síndrome de errores, la información aparece en forma de múltiples tablas desde la que se expresan los mismos datos considerando distintas clasificaciones: los lugares de inyección, los tipos de módulos, los tipos de fallo inyectados o de forma global.

Los datos calculados en este caso son bastante diferentes:

- Los porcentajes de errores activados, detectados y recuperados (por el sistema global y por cada mecanismo).
- Las latencias medias de propagación, detección y recuperación de errores.
- Los coeficientes de cobertura (tanto del sistema global como de cada mecanismo).

6.6 Situación actual

Hasta el momento, los avances conseguidos han sido:

- Generalizar el tratamiento de gran número de modelos en VHDL. El ASL es capaz de procesar la mayor parte de los elementos que pueden aparecer en un diseño en VHDL.

  Sin embargo, debido a la elevada complejidad de la sintaxis de este lenguaje de descripción de *hardware* (próximo a los lenguajes de programación de alto nivel en muchos aspectos), todavía no se ha alcanzado la generalización total, ya que aún no se procesan aspectos como el tratamiento de *alias* y grupos.

- Implementar completamente la técnica de inyección mediante órdenes del simulador. En particular, se pueden inyectar fallos simples utilizando un completo conjunto de modelos de fallos (*stuck-at* (‘0’, ‘1’), *bit-flip*, *pulse*, *indetermination*, *open-line* y *delay*). También se han hecho algunas pruebas de inyección de fallos múltiples [Baraza *et al.* 2002], si bien esta capacidad no está integrada todavía.

  Las otras dos técnicas (perturbadores y mutantes) no están integradas en la herramienta, si bien se han llevado a cabo experimentos conducentes a probar la inyección con dichas técnicas y a comparar los resultados obtenidos [Gracia *et al.* 2000, Gracia *et al.* 2001a, Gracia *et al.* 2001b]. Además, algunos elementos de la herramienta ya están pre-

parados para el manejo de dichas técnicas. Por ejemplo, el ASL es capaz de generar el árbol lexicográfico del modelo para la técnica de perturbadores, y con pequeños cambios también podría hacerlo para los mutantes.

- Paralelizar un experimento de inyección en tareas que se pueden ejecutar en ordenadores diferentes, llevándolo a cabo de manera en cierto modo distribuida, aun cuando la elección y distribución de las tareas estén todavía a cargo del usuario.

6.7 Comparación de VFIT con otras herramientas similares

Como se vio en el apartado 5.6, hay pocas herramientas de inyección de fallos mediante simulación de modelos en VHDL con las que se pueda comparar VFIT. Si se descartan las aportaciones que no presentan una herramienta, y aquéllas que implementan las denominadas "Otras técnicas", como es el caso de la herramienta descrita en [DeLong *et al.* 1994] y de VERIFY [Sieh *et al.* 1997], quedan sólo cuatro: MEFISTO [Jenn *et al.* 1994], MEFISTO-C [Folkesson *et al.* 1998], MEFISTO-L [Boué *et al.* 1998] y Fault Detector System [Corno *et al.* 2000].

En la Tabla 6.2 se muestra una comparación de las cinco herramientas, atendiendo a una serie de criterios:

- La plataforma donde se ejecuta.
- El nivel de abstracción de los modelos sobre los que se aplica.
- Las técnicas de inyección utilizables.
- Las clases de fallos que se pueden inyectar, atendiendo a su persistencia.
- El conjunto de modelos de fallos para cada técnica implementada.
- La forma de generar el instante de inyección.
- Si existe alguna clase de paralelismo en alguna de las fases del proceso de inyección de fallos.

Como se despende de la tabla, VFIT presenta tres importantes ventajas respecto a las otras cuatro herramientas. En primer lugar, permite aplicar las tres principales subtécnicas de inyección (*órdenes del simulador*, *perturbadores* y *mutantes*). También es posible inyectar fallos de cualquier persistencia (permanentes, intermitentes y transitorios). Por último, y más importante, el conjunto de modelos de fallos que se pueden inyectar es muy superior. También es de destacar el hecho de que VFIT se ejecuta en una plataforma muy común, mientras las demás herramientas funcionan en sistemas bajo UNIX.

Como principal desventaja, se puede observar una relativa falta de paralelismo en la realización de las campañas de inyección, que afecta principalmente a la duración de la fase de inyección.

		MEFISTO	MEFISTO-C	MEFISTO-L	Fault Detector System	VFIT
	Plataforma / Sist.Operativo	Red de Sun IPC / UNIX	Red de Sun Sparc / UNIX	Red de est. de trabajo / UNIX	Sun Ultra 5 / UNIX	**PC / Windows®**
	Nivel(es) de abstracción	Puerta RT Chip	Puerta RT Chip	Puerta RT Chip	RT	**Puerta RT Chip**
Técnicas implementadas	**Órdenes del simulador**	Sí	Sí	No	Sí	**Sí**
	Perturbadores	No	No	Sí	No	**En proceso[2]**
	Mutantes	Propuesta[1]	No	No	No	**En proceso[2]**
Modelos de fallo según técnica[3]	**Órdenes del simulador**	*Stuck-at* (‘0’, ‘1’), *Bit-flip*	*Stuck-at* (‘0’, ‘1’), *Bit-flip*	–	*Stuck-at* (‘0’, ‘1’), *Bit-flip*	***Stuck-at* (‘0’, ‘1’), *Bit-flip*, *Pulse*, *Indetermination*, *Open-line*, *Delay***
	Perturbadores	–	–	*Stuck-at* (‘0’, ‘1’), *Bit-flip, Open-line, Indetermination, Short, Bridging*	–	***Stuck-at* (‘0’, ‘1’), *Bit-flip*, *Pulse*, *Indetermination*, *Open-line*, *Delay*, *Stuck-open*, *Short*, *Bridging***
	Mutantes	Cambios sintácticos en el código VHDL [Armstrong *et al.* 1992]	–	–	–	**Cambios sintácticos en el código VHDL [Armstrong *et al.* 1992]**
Tipos de fallos (persistencia)	**Transitorios**	Sí	Sí	Sí	No	**Sí**
	Permanentes	Sí	Sí	Sí	Sí	**Sí**
	Intermitentes	No	No	No	No	**Sí**
	Instante de inyección	Tiempo (valor fijo o aleatorio –distr. uniforme), Palabra de disparo, Puntos de ruptura	Tiempo (aleatorio –distribución uniforme)	Tiempo (valor fijo o aleatorio –distr. uniforme), Palabra de disparo, Puntos de ruptura	Puntos de ruptura	**Tiempo (valor fijo o aleatorio –varias funciones de distribución)**
	Paralelismo	Inyección	Inyección	Inyección	No	**Inyección[4] Análisis[4]**

(1) Según [Leveugle y Hadjiat 2000] no está realmente implementada.
(2) Aunque se han publicado algunos resultados, estos son provisionales (a nivel de prototipo), ya que esta técnica todavía no está integrada en la herramienta.
(3) En todas las herramientas se consideran únicamente fallos simples, excepto en VFIT, donde se han realizado algunas pruebas [Baraza *et al.* 2002], aunque esta capacidad no está todavía integrada.
(4) De manera parcial, mediante sesiones. La distribución real está en estudio.

Tabla 6.2: Comparación de VFIT con otras herramientas de inyección de fallos mediante simulación de modelos en VHDL.

6.8 Resumen y conclusiones

En este capítulo se han descrito las características principales de VFIT, una herramienta de inyección de fallos mediante simulación de modelos en VHDL que se ha desarrollado como elemento central de este trabajo.

VFIT ha sido concebida para realizar campañas de inyección sobre modelos en VHDL de cualquier complejidad, en ordenadores personales o compatibles bajo entorno Windows®, usando como base un simulador de VHDL comercial (Modelsim® de Model Technology®).

Los experimentos de inyección se dividen en tres fases: configuración de los parámetros del experimento, simulación y análisis y extracción de resultados. La fase de configuración es la única en la que la herramienta precisa de interactuar con el usuario. Una vez configurado el experimento, VFIT ejecuta las otras dos fases de forma automática. Para optimizar el tiempo de simulación, a la sazón el mayor inconveniente de las técnicas de inyección basadas en simulación, se ha dotado a la herramienta de la capacidad de ejecutar sólo un subconjunto de las tareas implicadas en las fases de simulación y análisis (al que se denomina *sesión*): sólo la fase de simulación, sólo la fase de análisis, o las fases de simulación y análisis de una parte del número de total de fallos inyectados. Esta característica permite paralelizar (por el momento de manera manual) la realización de los experimentos de inyección, ejecutando sesiones diferentes en ordenadores distintos, reduciendo de este modo la duración total del experimento.

Con VFIT se pueden inyectar fallos utilizando las tres subtécnicas principales de simulación de modelos en VHDL abordadas en el presente trabajo de tesis: órdenes del simulador, perturbadores y mutantes. Además, además inyectar un completo conjunto de modelos de fallos permanentes, transitorios e intermitentes. Este conjunto abarca otros modelos de fallos además de los tradicionales *stuck-at* y *bit-flip*. Conviene destacar los modelos *indetermination*, *delay*, y la distinción entre los fallos *bit-flip* (aplicable a elementos de memoria) y *pulse* (para lógica combinacional).

En cuanto al instante y la duración de los fallos, es posible optar entre especificar valores concretos o generarlos aleatoriamente, pudiéndose elegir entre diferentes funciones de distribución: Uniforme, Exponencial, Normal y Weibull.

Un aspecto interesante de VFIT es el hecho de que, a partir de las trazas de simulación generadas en la fase de inyección, es posible realizar dos tipos de estudios del sistema: análisis del síndrome de error o validación de los mecanismos de tolerancia a fallos de un sistema tolerante a fallos.

Para mostrar la facilidad de manejo de VFIT se presentan dos ejemplos simples de cómo se realizarían el estudio del síndrome de error de un sistema y la validación de un sistema tolerante a fallos, haciendo hincapié en la interfaz con el usuario y en algunos de los ficheros resultantes en el proceso.

Por último, se ha comparado VFIT con otras herramientas de características similares: MEFISTO, MEFISTO-C, MEFISTO-L y Fault Detector System. Las principales diferencias radican en la plataforma utilizada (VFIT es la única que se ha diseñado para ordenadores personales) y en las técnicas de inyección y los modelos de fallos utilizadas.

6.9 Trabajo futuro

Futuras modificaciones previstas en VFIT son:

1. Integrar las técnicas de perturbadores y mutantes, resolviendo además los problemas inherentes a ésta última: la duración de la simulación y la sincronización.
2. Permitir la inyección de fallos múltiples, tanto desde el punto de vista temporal (ocurrencia de varios fallos en una simulación) como en el espacial (ocurrencia de varios fallos simultáneos debidos a una misma causa). Para ello hay que incluir nuevas *macros* en la librería de *macros* de inyección.
3. Permitir la inyección de los fallos de manera distribuida, ejecutando las *macros* de inyección en otras máquinas de manera remota, sin intervención del usuario en el reparto de las tareas.
4. Investigar e implementar otras técnicas para optimizar el coste temporal del proceso global: inyección-simulación-análisis.
5. Ampliar el conjunto de funciones de distribución de probabilidad para generar el instante de inyección y la duración del fallo. Algunas funciones interesantes son Lognormal, Rayleigh y Gamma (véase el apéndice A).

7 Experimentos de inyección realizados

7.1 Introducción

Con VFIT se han llevado a cabo numerosos experimentos sobre varios modelos diferentes, y con distintos objetivos. En este capítulo se presenta una selección de los más relevantes, incluyendo la descripción de los diferentes modelos sobre los que se ha aplicado VFIT:

- Un sistema no tolerante a fallos basado en el procesador MARK2.
- Un sistema tolerante a fallos basado en el procesador MARK2.
- El microcontrolador PIC.
- El microcontrolador 8051.
- El controlador de comunicaciones basado en la arquitectura TTA (Time-Triggered Architecture) TTP/C-C1.

Por otro lado, los experimentos que aquí se muestran se pueden clasificar, en función de su objetivo, en:

- **Experimentos de calibración de la herramienta**. Aquí se incluyen todos los experimentos realizados para comprobar el correcto funcionamiento de VFIT. Para ello se han repetido los principales experimentos que se llevaron a cabo con el prototipo inicial, consistentes en estudiar el síndrome de error del sistema MARK2 no tolerante a fallos y en validar el sistema MARK2 tolerante a fallos.
- **Experimentos de validación**. En este epígrafe se incluyen todos los experimentos realizados para validar un sistema tolerante a fallos. Estos experimentos se han llevado a cabo sobre dos modelos: el sistema MARK2 tolerante a fallos, y el controlador de comunicaciones TTP/C-C1.
- **Estudio de representatividad de los modelos de fallos a nivel RT**. Aquí se engloban una serie de experimentos, en principio orientados a estudio del síndrome de error de diversos modelos de procesador. En realidad están enfocados a analizar cómo se propagan los fallos desde el nivel lógico al RT, con el fin de establecer cuáles son los modelos de fallo más representativos en el nivel RT. Se mostrarán algunos de los resultados obtenidos a partir de la realización de estos experimentos sobre los modelos de los microcontroladores PIC y 8051.

La organización del capítulo es como sigue. En el apartado 7.2 se describen los diferentes modelos de sistemas sobre los que se ha aplicado VFIT. En los apartados 7.3 a 7.5 se enumeran y describen las campañas de inyección realizadas sobre dichos modelos, y se muestran algunos de los resultados más relevantes obtenidos de dichas campañas, así como las publicaciones a que han dado lugar dichos experimentos.

Para cada bloque de experimentos se indican los objetivos principales (especificando los resultados publicados), los parámetros de inyección (distinguiendo entre los parámetros que no varían en todos los experimentos y los que sí lo hacen) y los parámetros de análisis más relevantes, y se comentan los resultados más interesantes.

Posteriormente, en el apartado 7.6 se hace una recapitulación de los trabajos realizados, y en el 7.7 se indican algunas líneas de trabajo futuro e incluso algunas que ya se están desarrollando y que no han sido reflejadas en este trabajo.

7.2 Descripción de los modelos sobre los que se ha aplicado VFIT

7.2.1 Sistema no tolerante a fallos basado en el procesador MARK2

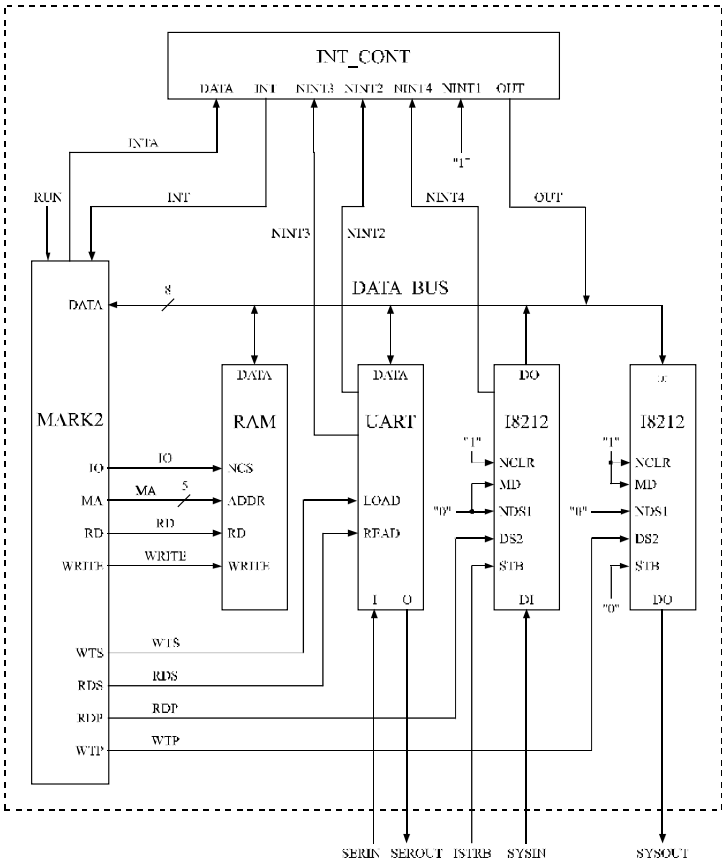

Figura 7.1: Diagrama de bloques del sistema basado en el procesador MARK2 [Gil *et al.* 1997b].

La Figura 7.1 muestra el diagrama de bloques del sistema no tolerante a fallos basado en el procesador de 8 bits MARK2 [Armstrong 1989]. La arquitectura estructural del modelo consta de los siguientes componentes:

- Microprocesador de 8 bits (`MARK2`).
- Memoria RAM (`RAM`).
- Puerto paralelo de entrada (`I8212`).
- Puerto paralelo de salida (`I8212`).
- Puerto serie-paralelo/paralelo-serie (`UART`).
- Controlador de interrupciones (`INT_CONT`).

Cada componente se modela con una arquitectura comportamental que consta habitualmente de uno o más procesos concurrentes.

La Tabla 7.1 resume el conjunto de instrucciones del procesador MARK2.

Instrucción	Código de operación	Descripción
JMP dir	000	Salto incondicional (absoluto): PC ← dir
TCA	001	Complemento a 2 del acumulador (ACC): ACC ← - ACC
LDA dir	010	Carga del acumulador desde la memoria: ACC ← (dir)
STA dir	011	Almacenamiento del acumulador en la memoria: (dir) ← ACC
ADD dir	100	Suma el dato direccionado en memoria al acumulador: ACC ← ACC + (dir)
INT	101	Control de interrupciones: Habilitación, deshabilitación, retorno.
JPN	110	Salto si el acumulador es negativo: Si ACC(7) = '1' entonces PC ← dir
STP	111	Parada

Tabla 7.1: Juego de instrucciones del procesador MARK2 [Gil 1999].

El modelo del procesador es sencillo, pero contiene los elementos básicos de la mayoría de los computadores de propósito general. Se trata de un modelo comportamental bastante concurrente, tal como refleja la Figura 7.2.

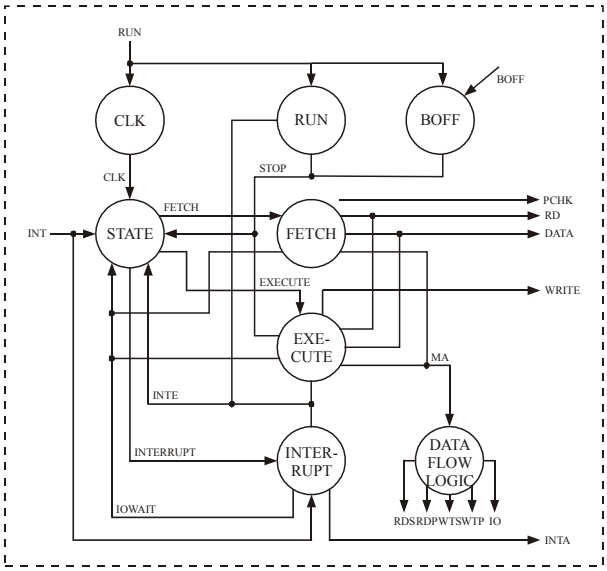

Figura 7.2: Grafo de procesos del procesador MARK2 [Gil *et al.* 1997b].

La Figura 7.2 muestra el grafo de procesos de la arquitectura comportamental del procesador, con las señales que comunican los procesos entre sí. Como puede observarse, consta de los siguientes procesos:

- `CLK`, encargado de generar el reloj del sistema.
- `RUN`, que controla el arranque y la parada del procesador.
- `FETCH`, que realiza el ciclo de búsqueda.
- `EXECUTE`, que lleva a cabo el ciclo de ejecución.
- `INTERRUPT`, encargado de manejar los ciclos de interrupción.
- `STATE`. Su misión es controlar la activación de los procesos `FETCH`, `EXECUTE` e `INTERRUPT`.
- `DATA_FLOW`. Representa la lógica necesaria para decodificar las señales de E/S.

La arquitectura comportamental de la memoria RAM consta de un único proceso, activado por las señales: `CS` (*chip select*), `Read` y `Write`. La memoria es una matriz bidimensional de 32 bytes, fácilmente ampliable.

7.2.2 Sistema tolerante a fallos basado en el procesador MARK2

Se ha diseñado el modelo VHDL de un sistema microcomputador tolerante a fallos. La Figura 7.3 muestra el diagrama de bloques del sistema. Se trata de un sistema duplicado con repuesto en frío (*duplex system with cold stand-by sparing*), detección de paridad y temporizador de guardia (*watchdog timer*).

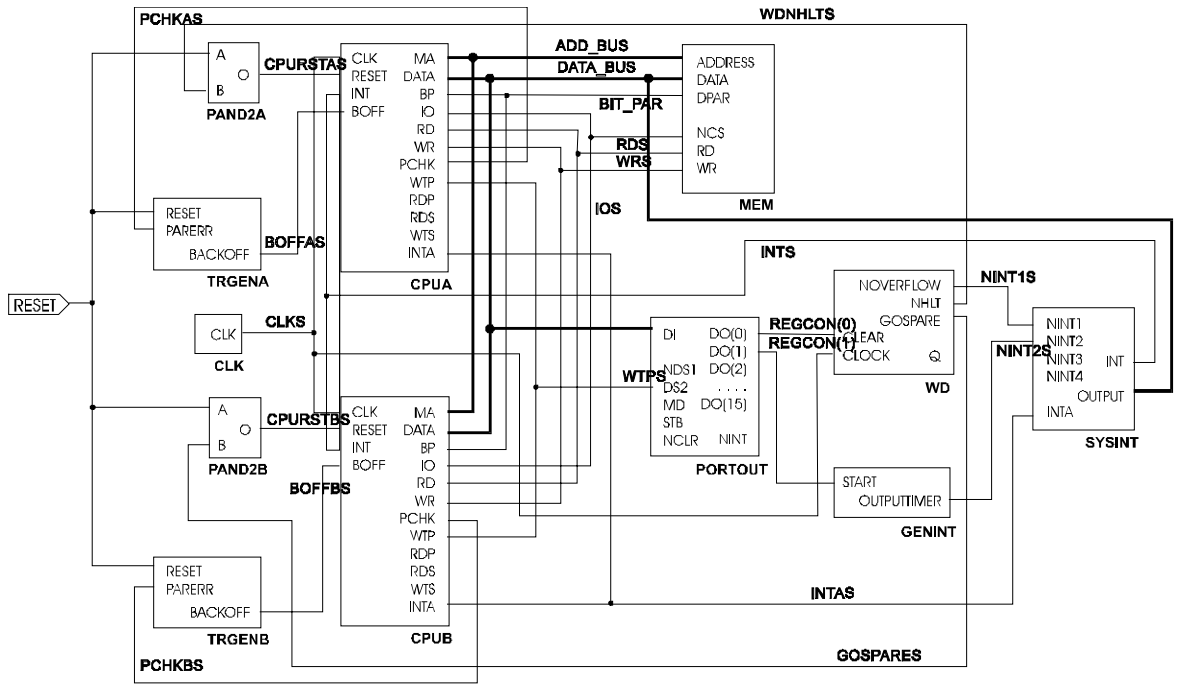

Figura 7.3: Diagrama de bloques del sistema tolerante a fallos basado en el procesador MARK2 [Gil 1999].

La arquitectura estructural del modelo consta de los siguientes componentes:

- Procesador principal (CPUA) y CPU de repuesto (CPUB).
- Memoria RAM (MEM).
- Puerto paralelo de salida (PORTOUT).
- Controlador de interrupciones (SYSINT).
- Generador de reloj (CLK).
- Temporizador de guardia (WD).
- Generador de pulso (GENINT).
- Dos generadores de ciclo de *Backoff* (TRGENA, TRGENB).
- Dos puertas AND (PAND2A, PAND2B).

Cada componente se modela con una arquitectura comportamental que consta habitualmente de uno o más procesos concurrentes. El tamaño de la memoria RAM se ha ampliado a 4 Kbytes (frente a los 32 bytes del modelo original) para poder almacenar programas de mayor complejidad. El procesador principal y el de repuesto son versiones mejoradas del procesador MARK2 descrito en el apartado anterior. Se han ampliado los buses de datos y direcciones (16 bits y 12 bits respectivamente). También se han añadido nuevos registros, instrucciones y modos de direccionamiento. Se ha mejorado el manejo de la pila y las interrupciones. En la Tabla 7.2 se puede observar el juego de instrucciones del nuevo procesador.

La instrucción LDA dir (incluida en el juego de instrucción del procesador MARK2, pero no en el la versión mejorada) se puede implementar con la secuencia:

LDI dir

LIR ACC

Además, se han incorporado algunas señales de control relacionadas con la detección de paridad (PCHK) y la generación de un ciclo de reintento[72] (BOFF). El bit de paridad se genera en el ciclo de escritura de la memoria, y se añade al bus de datos (señal BIT_PAR). La detección de paridad se efectúa en el ciclo de lectura de la memoria. En caso de error, se activa la señal PCHK.

La Figura 7.4 muestra el grafo de procesos del modelo comportamental del procesador, con las señales que comunican los procesos entre sí. Como puede observarse, consta de los siguientes procesos:

- Por un lado, CLK, RUN, FETCH, EXECUTE, INTERRUPT, STATE y DATA_FLOW, que realizan las mismas funciones que en la versión no tolerante a fallos del procesador.
- BACKOFF, encargado de realizar las tareas relacionadas con el ciclo de *backoff*.

[72] En inglés, *backoff cycle* (o también *retry cycle*).

Instrucción	Descripción
JMP dir	Salto incondicional (absoluto): PC ← dir
TCA	Complemento a 2 del acumulador (ACC): ACC ← - ACC
STA dir	Almacenamiento del acumulador en la memoria: (dir) ← ACC
ADD dir	Suma el dato direccionado en memoria al acumulador: ACC ← ACC + (dir)
JPN	Salto si el acumulador es negativo: Si ACC(15) = '1' entonces PC ← dir
HLT	Parada
PUSH reg	Apila registro reg: SP ← SP-1 (SP) ← reg
POP reg	Desapila registro reg: Reg ← (SP) SP ← SP-1
JSR dir	Llamada a subrutina: PC ← PC +1 PUSH PC PC ← dir
RET	Retorno de subrutina: POP PC
LDI dato	Carga inmediata del acumulador: ACC ← dato
LIR reg	Carga indirecta del acumulador: ACC ← (reg)
STI reg	Almacenamiento indirecto de registro: (ACC) ← reg
XCH reg	Intercambio de registros: ACC ↔ reg
INT dir	Llamada a interrupción: PC ← PC + 1 PUSH ACC PUSH PC PC ← dir
RTI	Retorno de interrupción: POP PC POP ACC

Tabla 7.2: Juego de instrucciones de la versión tolerante a fallos del procesador MARK2 [Gil 1999].

Como se mencionó anteriormente, se han añadido algunos mecanismos de tolerancia a fallos para aumentar la Confiabilidad del sistema. Los mecanismos de detección de errores incluyen la detección de paridad y el control de flujo de programa mediante un temporizador de guardia.

Si el procesador detecta un error de paridad, el generador del ciclo de *backoff* activa su señal durante un tiempo fijo (que puede ser programado). El procesador espera durante ese

tiempo a que finalice la causa del error, tras lo cual vuelve a ejecutar la última instrucción. En caso de que el error de paridad persista, la señal `Backoff` se activa de manera permanente.

Se ha utilizado una interrupción periódica (`NINT2`) para detectar los errores de control de flujo de programa (*Control Flow*). Cada vez que se recibe la interrupción, la rutina de respuesta reinicia el temporizador de guardia para evitar su desbordamiento. Seguidamente, el componente `GENINT` es activado para generar una nueva interrupción. Estas acciones se realizan a través del puerto paralelo de salida.

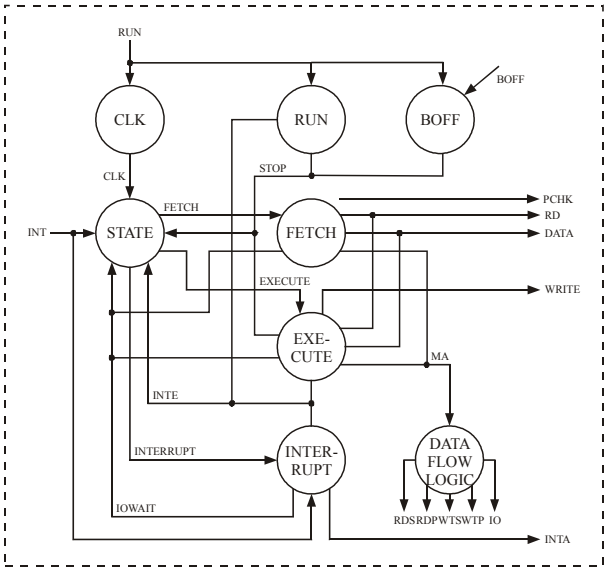

Figura 7.4: Grafo de procesos de la versión tolerante a fallos del procesador MARK2 [Gil 1999].

Un error de control de flujo de programa durante la rutina de interrupción producirá el desbordamiento del temporizador de guardia. Esta situación activa la señal de interrupción del procesador (`NINT1`) para recuperar el sistema desde un punto de recuperación (*Checkpoint*) previamente almacenado en memoria estable.

Hay un segundo banco con una copia de repuesto del punto de recuperación, y una variable que indica el banco activo. De esta manera se asegura la integridad de los datos en todo momento, y se configura un sistema de memoria estable.

En caso de que tengan lugar dos desbordamientos sucesivos del temporizador de guardia, la señal `NHLT` se activa para parar permanentemente el procesador principal y activar el procesador de repuesto. El procesador de repuesto lee el punto de recuperación de la memoria estable y continúa la tarea del procesador principal.

7.2.3 Microcontrolador PIC

El PIC es un microcontrolador de 8 bits desarrollado por Microchip [PIC16C5X 2003]. El modelo en VHDL que se ha utilizado está inspirado en el modelo sintetizable denominado PIC16C5X, realizado por Ernesto Romani [PIC16C5X 1998]. Sobre este modelo base se han realizado algunas modificaciones para mejorar algunos aspectos de cara a su utilización para la inyección de fallos con VFIT.

El modelo del PIC16C5X consta de dos componentes principales: la unidad de control y la unidad de procesamiento de datos (*datapath*). Todo el microcontrolador está gobernado por una única señal de reloj (externa).

La unidad de control (encargada de la interpretación de las instrucciones y de la generación de las señales de control necesarias para ejecutar cada instrucción) está implementada de forma comportamental, mediante dos procesos. Uno de ellos se dedica a la activación de las señales de control de los registros; el otro, a la generación del resto de las señales de control.

Por su parte, la unidad de procesamiento lleva a cabo las acciones especificadas por la unidad de control. Está implementada de mediante una descripción estructural, y sus componentes son los elementos del microcontrolador que elaboran y/o almacenan información, y están conectados a través de buses. La Figura 7.5 muestra la estructura de la unidad de procesamiento del PIC16C5X.

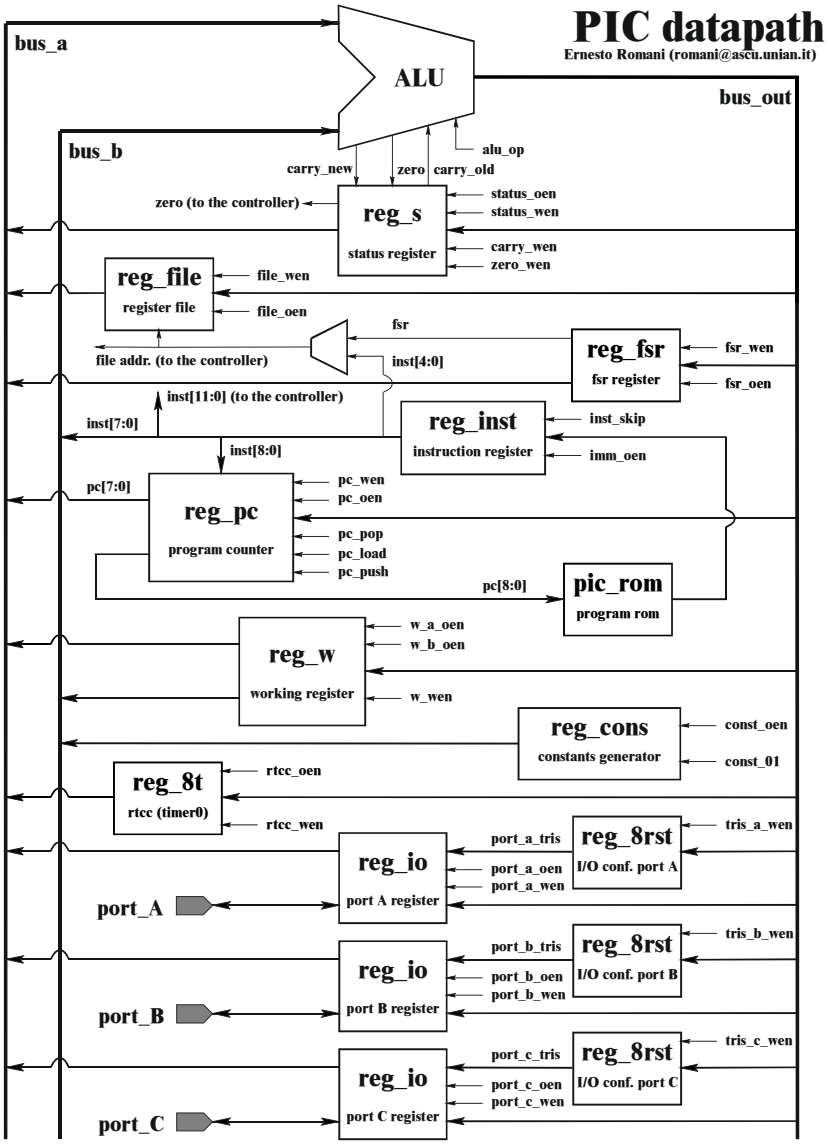

Figura 7.5: Diagrama de bloques de la unidad de procesamiento (*datapath*) del modelo del microcontrolador PIC16C5X [PIC16C5X 1998].

El componente principal de la unidad de procesamiento es la `ALU`, que realiza las operaciones aritméticas y lógicas. Dos buses tri-estado (`bus_a` y `bus_b`) llevan los operandos a la `ALU`. Un tercer bus, llamado `bus_out` transporta la salida de la `ALU`. Casi todos los movimientos de datos se llevan a cabo mediante estos tres buses.

El contador de programa (`reg_pc`), de 9 bits, permite direccionar la memoria ROM interna de programa (`pic_rom`) de hasta 512×12 bits.

Gracias a la existencia de un bus separado para el código (implementado por la señal `inst`), la fase de búsqueda de la instrucción (*fetch*) se realiza en paralelo con las fases de decodificación y ejecución. Por ello, las instrucciones se ejecutan en un solo ciclo de reloj.

En el mismo componente donde se implementa el contador de programa (`reg_pc`) se implementa también una pila *hardware* de 9 bits de dos niveles (aunque la profundidad de la pila se puede modificar fácilmente).

El tamaño del banco de registros es fácilmente modificable. Los registros se dividen en 8 de función específica[73], y el resto son de propósito general. En cuanto a su implementación en el modelo, los de función específica (TMR0, PC –los 8 bits de menor peso–, STATUS y FSR) están realizados como diseños individuales (`rtcc`, `program_counter`, `status_register`, y `fsr_register`), mientras que los de propósito general están implementados en un único componente (`register_file`). Al temporizador `rtcc` no se le ha incorporado ninguna función especial, y puede emplearse como un registro de propósito general.

Tiene implementados tres puertos de E/S de 8 bits, (`port_a_register`, `port_b_register` y `port_c_register`), y no tiene implementado el temporizador de guardia (*watchdog timer*).

El juego de instrucciones soportado coincide con el del microcontrolador original excepto en tres instrucciones que no han sido incorporadas: OPTION, SLEEP y CLRWDT. Estas instrucciones están relacionadas con elementos del microcontrolador real no implementados en el modelo en VHDL (el temporizador de guardia y un registro de configuración asociado a éste).

Sobre el modelo original se han realizado diversos cambios para favorecer la inyección de fallos. Por ejemplo, para poder inyectar fallos de tipo *delay* en la señal de reloj del sistema, se ha declarado una constante genérica. También se ha implementado una versión estructural de la ALU (para poder inyectar fallos en la lógica combinacional –véase el apartado 7.5.1) y se han diseñado varias versiones de la memoria ROM para poder ejecutar diferentes cargas de prueba (*workload*).

7.2.4 Microcontrolador 8051

El 8051 es un microcontrolador de 8 bits desarrollado originalmente por Intel en 1980. El modelo en VHDL que se ha utilizado está inspirado en el modelo sintetizable del MC8051 realizado por Oregano Systems [MC8051 2002]. Para optimizar algunos aspectos al inyectarle fallos con VFIT se han realizado algunas modificaciones sobre el modelo inicial.

[73] SFR,del inglés *special function registers*.

Como se puede observar en la Figura 7.6, el modelo del MC8051 está compuesto por la CPU (`mc8051_core`) y tres módulos de memoria: 128 bytes de RAM interna (`mc8051_ram`), hasta 64Kbytes de ROM interna (`mc8051_rom`) y hasta 64Kbytes de RAM externa (`mc8051_ramx`). El núcleo del microcontrolador en sí está compuesto por la unidad de control (`mc8051_control`), la ALU (`mc8051_alu`) y un número configurable de interfaces serie (`mc8051_siu`) y de temporizadores/contadores (`mc8051_tmrctr`). Además, tiene cuatro puertos paralelos no multiplexados.

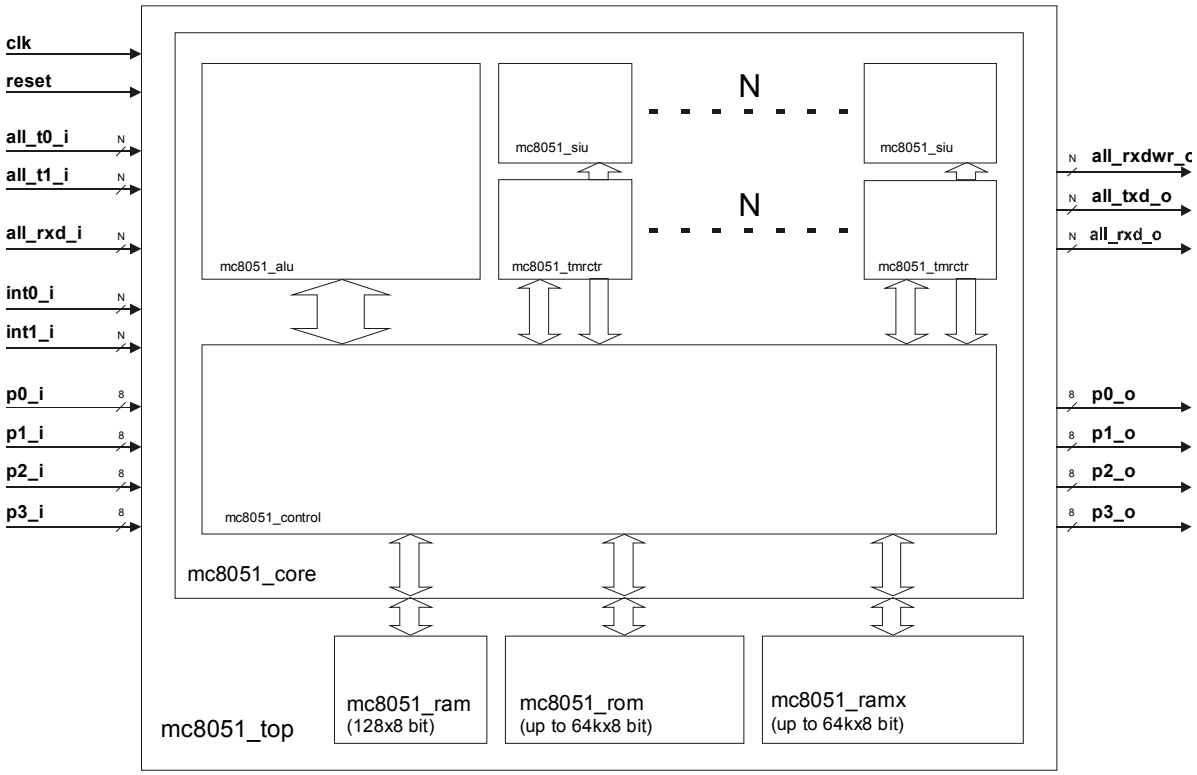

Figura 7.6: Diagrama de bloques del modelo del microcontrolador MC8051 [MC8051 2002].

Por otra parte, la ALU puede configurarse para dotarle de multiplicador y divisor. Al igual que se hizo con el modelo del microcontrolador PIC16C5X, se ha modificado la implementación de la ALU, haciéndola estructural con el objeto de poder inyectar fallos en la lógica combinacional (véase el apartado 7.5.2).

7.2.5 Controlador de comunicaciones TTP/C-C1

Este controlador está basado en la arquitectura TTA (*Time-Triggered Architecture*) [Kopetz 1998], que implementa TTP (*Time-Triggered Protocol*), un protocolo de comunicaciones tolerante a fallos para sistemas distribuidos de tiempo real. El protocolo TTP es síncrono, y tiene una planificación estática y cíclica. La principal característica del protocolo es el silencio ante avería (en inglés *fail silence*), que asegura que cualquier nodo funciona correctamente o dejará de comunicarse.

Para el intercambio de mensajes entre nodos, el protocolo emplea el esquema TDMA (*Time Division Multiple Access*), en el que el responsable del inicio de una comunicación es el tiempo.

Para comprender cómo funciona el protocolo TTP, es preciso definir algunos conceptos:

- SRU (*Smallest Replaceable Unit*): Es un módulo conectado a un canal TTP™/C[74], cuya estructura se muestra en la Figura 7.7.

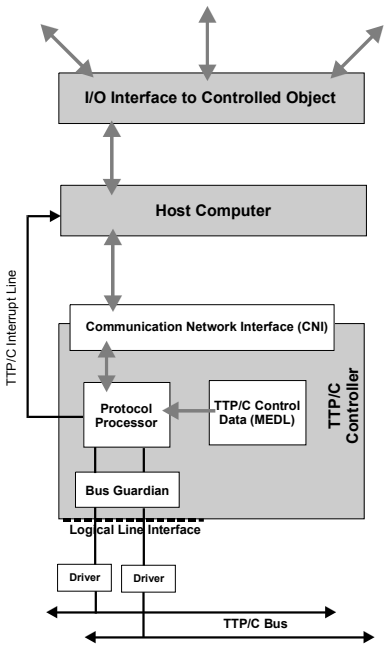

Figura 7.7: Diagrama de bloques de un SRU [TTP/C C1 2002].

- FTU (*Fault-Tolerant Unit*): Es un grupo de SRU que funcionan correctamente, incluso en caso de avería en una de ellas.
- *Slot*: Es el intervalo de tiempo más pequeño de una ronda de TDMA.
- SRU *slot*: Un *slot* asignado a una SRU.
- Ronda de TDMA: Secuencia de SRU *slots* de un ciclo del *cluster*.
- Ciclo de *cluster*: Secuencia de diferentes rondas de TDMA.

En la Figura 7.8 se muestran las características temporales de la transmisión en los sistemas basados en la arquitectura TTA.

La Figura 7.9 se puede apreciar el diagrama de bloques del controlador TTP™/C-C1[75], desarrollado por TTTech [TTP/C-C1 2002]. Este controlador de 16 bits está organizado en torno a la PCU (*Protocol Control Unit*). El diagrama de bloques lo completan algunos bloques funcionales de bajo nivel que implementan tareas críticas relacionadas con el protocolo: cálculo del CRC, transmisión de tramas, etc. La PCU controla la interacción entre los diferentes bloques funcionales de bajo nivel, y ejecuta mecanismos de alto nivel del protocolo, como el manejo de la redundancia, el servicio de pertenencia, etc. La PCU está implementada como un procesador segmentado con memoria interna de código y una ALU con acumulador. La carga de los programas en la memoria de la PCU se lleva a cabo desde una memoria EPROM, a

[74] La C es porque cumple los requisitos del estándar de seguridad de la industria automovilística SAE Class C [Class C 1994].
[75] C1 hace referencia a que es la primera versión del controlador.

través del *bus de registros*. Se trata de un bus interno síncrono, gobernado por la PCU, que conecta todos los bloques funcionales de bajo nivel.

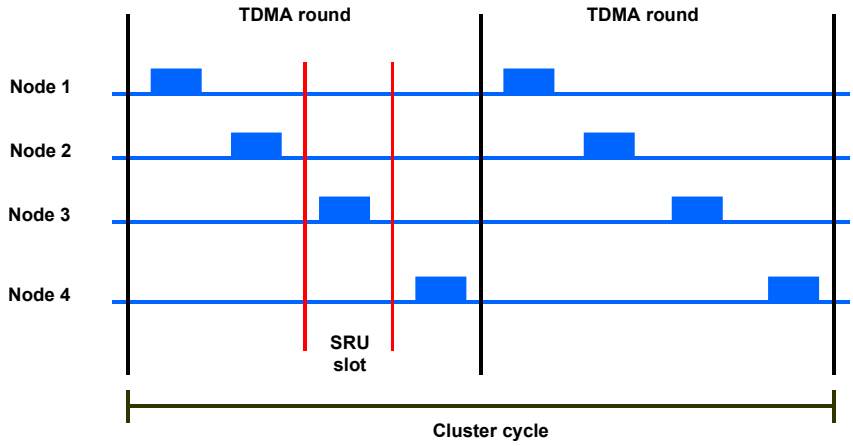

Figura 7.8: Diagrama de tiempos de la transmisión TTP™/C.

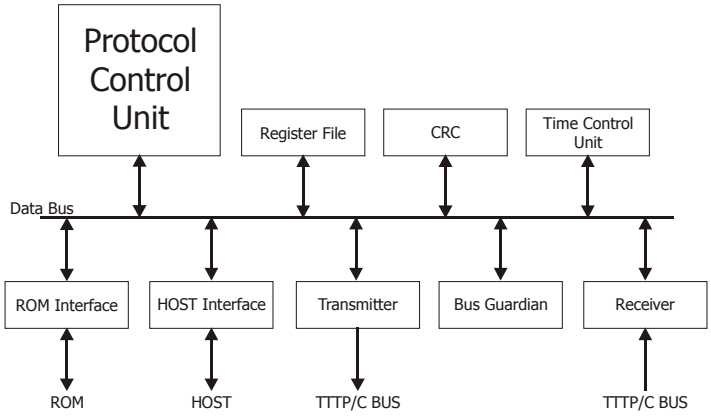

Figura 7.9: Diagrama de bloques del controlador TTP™/C-C1 [TTP/C-C1 2001].

La comunicación entre el controlador y el *Host Computer* se realiza a través de la CNI (*Communications Network Interface*, véase la Figura 7.7), implementada en el *Host Interface* como una memoria RAM de doble puerto. La CNI se divide en dos áreas, una de estado/control y otra de mensajes. El *host* utiliza la CNI para monitorizar al controlador, mientras que el controlador la emplea para almacenar mensajes (tanto de entrada como de salida) e información de control.

Otro componente importante es la MEDL (*Message Descriptor List*). Define el punto específico en el que transmitir una trama, según la planificación de comunicaciones predefinida. La MEDL se define en el arranque del sistema, y no es posible cambiarla en tiempo de ejecución. Se almacena en una memoria *flash* EPROM a la que el controlador de comunicaciones tiene acceso.

En cada transmisión, el emisor genera el CRC del mensaje incluyendo el estado de controlador (o *C-state*). El *C-state* está compuesto por el tiempo global del sistema, la posición actual en la MEDL, el modo de operación actual, los cambios de estado pendientes y la lista actual de pertenencias (o lo que es lo mismo, la situación de pertenencia o no al *cluster* de

cada nodo). Este estado debería ser reconocido por todos los nodos receptores, y en caso contrario, se produciría una discrepancia en la comunicación. En este caso, el nodo emisor sería marcado como "no perteneciente" al *cluster*, o lo que es lo mismo, habría perdido su pertenencia.

El controlador TTP™/C implementa cuatro mecanismos de detección de errores (EDM, del inglés *Error Detection Mechanisms*) [Kopetz 1999]:

- HE (*Host Error*): Su misión es detectar errores en el *host*. Cuando ocurre, el controlador entra en un estado pasivo, para cumplir con la especificación de silencio tras avería. Las causas de este error pueden ser un error de violación de modo[76] (el *host* solicita un cambio de modo no permitido), un error de *slot* ocupado[77] (el *slot* de emisión está ocupado por otro controlador), o un error de protocolo de escritura sin bloqueo[78].
- PE (*Protocol Error*): Detecta errores internos de protocolo del TTP™/C. En este caso, la ejecución del protocolo se suspende hasta que el *host* vuelve a poner en marcha el controlador. Este error puede estar provocado por distintas causas: un error de reconocimiento[79] (el controlador discrepa con la mayoría del *cluster*), un error de pertenencia[80] (se ha alcanzado el máximo número de errores de pertenencia consecutivos), caída del sistema de comunicaciones[81] (la única actividad en el bus detectada en la ronda de TDMA ha sido la propia), error de sincronización (en inglés, *Synchronisation Error*), error del guardián del bus (en inglés, *Bus Guardian Error*), error de CRC de la MEDL (en inglés, *MEDL CRC Error*) y descarga completada (en inglés, *Download Completed*).
- ML (*Membership Loss*): Detecta que el controlador ha perdido su pertenencia al *cluster*.
- BE (*Built-in self-test*): Éste no es realmente un mecanismo, sino un conjunto de mecanismos [Steininger y Temple 1999]:
  - ⇨ Concordancia de *C-state* (*C-state agreement*): Cuando un nodo recibe un mensaje, calcula el CRC incluyendo su propio *C-state*. Si el CRC no concuerda con el recibido, marca el mensaje como inválido y actualiza su estado para indicar la pérdida de pertenencia al *cluster* del nodo emisor. Este mecanismo se considera como un reconocimiento negativo para el emisor, tras el cual, su controlador de comunicaciones queda en silencio (se desconecta).
  - ⇨ Señal de vida del *host* (*host life sign*): Este mecanismo permite 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 queda en silencio.
  - ⇨ CRC del nivel de aplicación (*end-to-end CRC*): Como ya se ha comentado, se utiliza un CRC para proteger los datos. El controlador de comunicaciones considera al CRC como parte del mensaje.

[76] En inglés, *Mode Violation Error*.
[77] En inglés, *Occupied Slot Error*.
[78] En inglés, *Non-Blocking Write Protocol Error*.
[79] En inglés, *Acknowledgement Error*.
[80] En inglés, *Membership Error*.
[81] En inglés, *Communication System Blackout*.

- ⇨ Guardian de bus (*bus guardian*): Permite la transmisión de tramas en instantes de tiempo predefinidos, protegiendo la comunicación de los nodos con fallos. Para evitar fallos de modo común con el reloj del sistema, utiliza un reloj independiente.
- ⇨ Comprobación del CRC de la MEDL (*MEDL CRC check*): El controlador de comunicaciones también utiliza la unidad CRC para validar las entradas de la MEDL.

Por el contrario, el controlador de comunicaciones no dispone de mecanismos de recuperación de errores, porque se implementan a nivel superior: en el *Host Controller*.

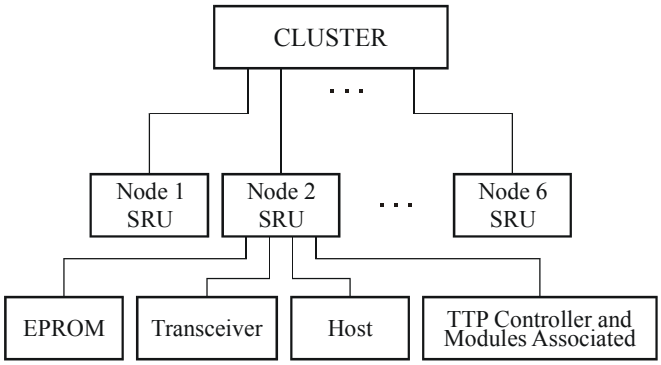

Figura 7.10: Diagrama de bloques del modelo VHDL del sistema TTP/C simulado.

El modelo en VHDL del sistema, también realizado por TTTech, es híbrido. La Figura 7.10 muestra el diagrama de bloques del modelo. Para los experimentos de inyección se ha simulado un *cluster* con seis nodos. Cada nodo se compone de un modelo comportamental de una memoria `EPROM` (que almacena el código del protocolo), un `Transceiver`, un `Host` y un modelo sintetizable del controlador TTP™/C-C1 y otros bloques funcionales de bajo nivel. Los modelos del microcontrolador y de los módulos asociados tienen una arquitectura estructural, con descripciones comportamentales en los componentes del nivel inferior. Las descripciones de los módulos sintetizables están implementadas a nivel RT. En cuanto a las cargas de prueba que ejecutan los nodos, se trata de aplicaciones sin ninguna función particular, salvo la de probar todas partes del modelo.

Los valores de los parámetros temporales del modelo son:

- El ciclo de *cluster* está compuesto por dos rondas de TDMA, y su duración es de 3.2 ms.
- Cada ronda de TDMA dura 1.6 ms, y está dividida en 6 SRU *slots*.
- Cada SRU *slot* dura aproximadamente 165 μs.

7.3 Experimentos de calibración de la herramienta

En este bloque se incluye una serie de experimentos que fueron originalmente realizados con el prototipo preliminar de la herramienta de inyección de fallos, y que después se repitieron con VFIT, a fin de comprobar el correcto funcionamiento de los nuevos algoritmos y funciones implementados en el nuevo prototipo. Los experimentos originales descritos en este apartado formaban parte del trabajo desarrollado por D. Daniel Gil para la realización de su tesis doctoral [Gil 1999].

Como ya se mencionó en el apartado 7.1, los experimentos que aquí se exponen se aplicaron sobre dos modelos: el sistema MARK2 (no tolerante a fallos) y la versión tolerante a fallos del mismo.

El apartado se divide en dos subapartados, uno por cada modelo. Para cada uno de ellos se incluyen una serie de subapartados en los cuales se describen los objetivos generales perseguidos por el conjunto de las campañas de inyección, los parámetros utilizados en las diferentes campañas, algunos de los parámetros de los estudios (análisis) realizados, y se comentan algunos de los resultados obtenidos.

Dado que estos experimentos se realizaron inicialmente con el prototipo preliminar de la herramienta, el autor se remite a la tesis doctoral de D. Daniel Gil [Gil 1999] para una explicación más extensa de los estudios realizados y los resultados obtenidos.

7.3.1 Experimentos realizados sobre el sistema MARK2

7.3.1.1 Objetivos

Con este grupo de experimentos se quería estudiar el síndrome de error del mencionado sistema para cubrir dos objetivos. Por un lado, se pretendía probar la validez de la técnica de inyección de fallos mediante simulación de modelos en VHDL. En segundo lugar, se quería efectuar una clasificación de los errores más habituales en el sistema, y calcular sus latencias de propagación. De esta forma, se podrían elegir los mecanismos de detección y recuperación de errores más adecuados para aumentar la Confiabilidad del sistema. Estos mecanismos se introdujeron en una segunda fase, en la que se diseñó la versión tolerante a fallos del sistema (véase su descripción en el apartado 7.2.2).

El estudio del síndrome de error contemplaba, además del efecto de los fallos en el sistema, la influencia de otros factores, como el tipo de lugar de inyección, la duración de los fallos, la función de distribución utilizada para generar el instante de inyección, o la carga de trabajo (*workload*) que ejecuta el sistema.

Fruto del trabajo desarrollado fueron las siguientes publicaciones: [Gil *et al.* 1997b, Gil *et al.* 1997c, Gil *et al.* 1998b, Gil *et al.* 1998c, Gil *et al.* 1998d].

7.3.1.2 Parámetros de inyección

Como ya se indicó en el apartado 7.1, los parámetros se pueden agrupar en dos clases. Por un lado, están los que son comunes a todos los experimentos. Por otro, aquéllos que han sido modificados para analizar algún aspecto particular.

Los parámetros comunes a todos los experimentos son:

- **Técnica de inyección aplicada**: Órdenes del simulador.
- **Número de fallos inyectados**: 3000[82].
- **Período de la señal de reloj (T)**: 1000 ns (1 μs).

[82] Realmente, en los experimentos de inyección cuyos resultados se publicaron en [Gil *et al.* 1997a, Gil *et al.* 1997b] sólo se inyectaron 193 fallos por experimento, por lo que no se incluyen en este resumen. Posteriormente se inyectaron 3000 fallos por experimento, lo que era suficiente para conseguir intervalos de confianza aceptables [Gil 1999].

- **Carga de trabajo**: Cálculo de la serie aritmética de *n* enteros, con *n* = 5.
- **Duración de la carga de trabajo (T_W)**: 227 μs.
- **Duración de la simulación**[83] **(T_S)**: 500 μs.
- **Lugares de inyección**: Se pueden inyectar fallos en todas las señales y variables atómicas del modelo.
- **Modelos de fallos**: *Stuck-at* ('0', '1') y *open-line*[84]. Por cada fallo inyectado, el modelo se selecciona de manera uniforme.

Los parámetros que han variado en alguno de los experimentos son:

- **Instante de inyección**: Se genera de manera aleatoria, siguiendo distintas funciones de distribución, y variando algunos parámetros en cada una:
  - ⇨ Uniforme, entre 0.0 ns y T_W (227 μs).
  - ⇨ Exponencial, con diferentes valores de α (0.002, 0.003, 0.004 y 0.005), con un factor de desplazamiento de 0.0, y con T_W como límite superior.
  - ⇨ Weibull, con diferentes valores de α (0.001 y 0.003) y β (0.2, 0.3, 0.4, 0.6, 0.8, 1.0, 2.0 y 3.0), con un factor de desplazamiento de 0.0, y con T_W como límite superior.
- **Duración de los fallos**: Se han inyectado fallos transitorios de duración fija, con los siguientes valores: 0.1T, 0.2T, 0.3T, 0.4T, 0.5T, 1.0T, 1.5T y 2.0 T, donde T es el período de la señal de reloj.

7.3.1.3 Parámetros de análisis

En todos los experimentos se han considerado los mismos parámetros:

- **Lugares donde se detectan los errores**: La propagación de los errores se detecta en las señales del nivel superior de la jerarquía de diseño utilizadas para interconectar los componentes. Éstas se corresponden fundamentalmente con el bus de direcciones, el bus de datos, y las señales de control de lectura/escritura.
- **Clasificación de los fallos**: Los lugares de inyección seleccionados en los parámetros de inyección se clasifican en función de su tipo, distinguiendo además entre los fallos inyectados en señales y variables. Se definen las siguientes clases de fallos:
  - ⇨ En señales:
    - ❑ Latch. Engloba los fallos inyectados en el registro de instrucciones (`IR`) y en el contador de programa (`PC`).

[83] Generalmente, la duración real de la simulación incluye tanto la duración de la carga de trabajo como un tiempo de observación posterior a la finalización de la carga, durante el cual el sistema todavía puede reaccionar ante los fallos insertados.

[84] Los modelos de fallos que se pueden aplicar están limitados a los tipos de datos de las señales y variables del modelo del sistema. Además, recuérdese que los parámetros descritos pertenecen a experimentos realizados con la versión preliminar de la herramienta de inyección (y empleando otro simulador). Por entonces, los modelos de fallos *indetermination*, *bit-flip* y *delay* no se podían implementar.

- Buses. Aquí se incluyen los fallos inyectados en el bus de direcciones (MA) y en bus de datos (DATA).
- Memory control. Son los fallos inyectados en las líneas de control de lectura/escritura de la memoria (RD y WR)
- Interrupt, que incluye los fallos en las señales INT e IOWAIT.
- Halt: fallos en las señales RUN y STOP.
- Clk, que contempla los fallos inyectados en el reloj del sistema (CLK).

⇨ En variables:

- ACC, que incluye los fallos inyectados en el acumulador.
- Mem: fallos inyectados en la memoria RAM.

- **Clasificación de los errores**: Se consideran tres clases de errores:

⇨ Control flow. Son errores que causan un cambio en el secuenciamiento de las instrucciones que no corresponde a ninguno de los caminos válidos previstos en la ejecución de la instrucción anterior. Están provocados por fallos en el bus de direcciones durante la fase de búsqueda, en el código de operación de una instrucción o en el bus de direcciones durante la ejecución de una instrucción JMP <dir>.

⇨ Data. Causan el acceso a una dirección errónea en una instrucción de lectura o escritura en memoria (LDA o STA), o una escritura de dato incorrecto en una instrucción en memoria (STA).

⇨ Others. Contempla los errores que no pertenecen a los dos grupos anteriores. Ejemplos de fallos causantes son fallos en saltos condicionales (JPN) que provocan el cambio del flujo ejecución hacia otro camino válido, y fallos que propician una parada del procesador (afectan a la instrucción STP y a la señal RUN).

7.3.1.4 Resultados

Entre otros estudios realizados, se han analizado [Gil *et al.* 1998b, Gil *et al.* 1998c]:

- Los porcentajes de errores efectivos (en conjunto y por cada clase de error) y sus latencias de propagación (distinguiendo entre señales y variables). Un error se considera efectivo cuando ocasiona una avería.
- La influencia del lugar de inyección en el porcentaje de errores efectivos, el tipo de error producido, y las latencias de propagación.
- La influencia de la duración de los fallos en el porcentaje de errores efectivos y las latencias de propagación.
- La influencia de la función de distribución en el porcentaje de errores efectivos y las latencias de propagación.

Las conclusiones extraídas a partir del estudio de los datos obtenidos se utilizaron para la realización de una versión tolerante a fallos del sistema, cuyo modelo se describe en el apartado 7.2.2)

7.3.2 Experimentos realizados sobre el sistema MARK2 tolerante a fallos

7.3.2.1 Objetivos

La intención de este primer bloque de experimentos sobre la versión tolerante a fallos del sistema MARK2, junto con los que se describen en el apartado 7.4.1, era validar la eficacia de los mecanismos de tolerancia a fallos insertados a partir de los resultados obtenidos del análisis del síndrome de error del sistema original.

En el estudio se incluyen aspectos como la influencia de la duración de los fallos o de la función de distribución utilizada para generar el instante de inyección.

Los resultados que aquí se exponen fueron publicados en [Gil *et al.* 1999].

7.3.2.2 Parámetros de inyección

Los parámetros comunes a todos los experimentos son:

- **Técnica de inyección aplicada**: Órdenes del simulador.
- **Número de fallos inyectados**: 3000.
- **Período de la señal de reloj (T)**: 1000 ns (1μs).
- **Carga de trabajo**: Cálculo de la serie aritmética de *n* enteros, con *n* = 5.
- **Duración de la carga de trabajo (T_W)**: 227 μs.
- **Duración de la simulación (T_S)**: 500 μs.
- **Lugares de inyección**: Se pueden inyectar fallos en todas las señales y variables atómicas del modelo excepto las de la CPU de repuesto (CPUB).
- **Modelos de fallos**: *Stuck-at* ('0', '1'), *indetermination*[85] y *open-line*. Por cada fallo inyectado, el modelo se selecciona de manera uniforme.

Los parámetros que han variado en alguno de los experimentos son:

- **Instante de inyección**: Se genera de manera aleatoria, siguiendo distintas funciones de distribución, y variando algunos parámetros en cada una:
  - ⇨ Uniforme, entre 0.0 ns y T_W (227 μs).
  - ⇨ Exponencial, con α = 0.005, con un factor de desplazamiento de 0.0, y con T_W como límite superior.
  - ⇨ Weibull, con α = 0.001, β = 3.0, con un factor de desplazamiento de 0.0, y con T_W como límite superior.
  - ⇨ Normal (o Gaussiana) con $\mu = T_W/2$, $\sigma = T_W/8$, con un factor de desplazamiento de 0.0, y con T_W como límite superior.

[85] En este modelo, los tipos de datos de las señales y variables permitían inyectar fallos *indetermination*. Sin embargo, a causa de la versión del simulador utilizada (en los experimentos originales) todavía no se podían implementar los modelos *bit-flip* y *delay*.

- **Duración de los fallos**: Se han inyectado fallos transitorios generados aleatoriamente con una distribución Uniforme entre 0.1T y 10.0T y transitorios de duración fija, con los siguientes valores: 0.1T, 1.0T, 10.0T, 100.0T, 250.0T y 500.0T, donde T es el período de la señal de reloj. Los dos últimos valores se pueden considerar como correspondientes a fallos permanentes, dada su elevada duración.

7.3.2.3 Parámetros de análisis

En todos los experimentos se han considerado los mismos parámetros:

- **Mecanismos de detección**: Paridad en las CPU (Parity) y el temporizador de guardia (Watchdog).
- **Mecanismos de recuperación**: Ciclo de reintento en las CPU (Backoff), puntos de comprobación (Checkpoint) y el repuesto en frío (Spare).

7.3.2.4 Resultados

En cada experimento se han calculado las siguientes medidas de la Confiabilidad (descritas con detalle en el apartado 6.3.6.1): el porcentaje de errores activados y sus latencias de propagación, las coberturas de detección y recuperación de errores (tanto de los mecanismos como globales), y las latencias de detección y recuperación de los errores. También se ha cumplimentado el grafo de predicados de los mecanismos de tolerancia a fallos (véase la Figura 6.4).

Comparando los resultados obtenidos en cada experimento, se analizó la influencia de la duración de los fallos y de la función de distribución utilizada para generar el instante de inyección en el porcentaje de errores activados, las latencias de propagación, detección y recuperación de errores y las coberturas de detección y recuperación.

7.4 Experimentos de validación

Los experimentos que aquí se destacan ya se realizaron con VFIT. Este hecho lleva emparejada la inclusión de nuevos modelos de fallos (particularmente *delay*, *bit-flip* y *pulse*).

Hasta ahora se han validado dos sistemas tolerantes a fallos: el basado en el microprocesador MARK2 (continuación de los experimentos descritos en el apartado 7.3.2) y el controlador de comunicaciones TTP/C-C1.

7.4.1 Experimentos realizados sobre el sistema MARK2 tolerante a fallos

7.4.1.1 Objetivos

El objetivo principal de estos experimentos ha sido, por un lado, ampliar la validación del sistema tolerante a fallos desarrollado. Por otro, estudiar la influencia de algunos de los factores determinantes para la inyección de fallos, como la técnica de inyección utilizada, los modelos de fallos, la duración de los fallos, la carga de trabajo, etc.

A consecuencia del trabajo desarrollado se han realizado las siguientes publicaciones: [Gracia *et al.* 2000, Gil *et al.* 2000, Baraza *et al.* 2000, Gracia *et al.* 2001a, Gracia *et al.* 2001b, Baraza *et al.* 2002, Gracia *et al.* 2002a, Gil *et al.* 2003a, Gil *et al.* 2003b].

7.4.1.2 Parámetros de inyección

Los parámetros comunes a todos los experimentos son:

- **Número de fallos inyectados**: 3000.
- **Período de la señal de reloj (T)**: 1000 ns (1μs).
- **Lugares de inyección**: Se pueden inyectar fallos en todas las señales y variables del modelo excepto las de la CPU de repuesto (CPUB).

Los parámetros que han variado en alguno de los experimentos son:

- **Técnicas de inyección aplicadas**: Órdenes del simulador, Perturbadores y Mutantes.
- **Cargas de prueba**:
  - ⇨ Cálculo de la serie aritmética de *n* enteros, con *n* = 6.
  - ⇨ Algoritmo de ordenación de la burbuja (*bubblesort*) para *n* enteros, con *n* = 6.
- **Duración de la carga de trabajo (T_W)**: Depende de la carga de trabajo:
  - ⇨ 264 μs para la serie aritmética.
  - ⇨ 884 μs para *bubblesort*.
- **Duración de la simulación (T_S)**: También es dependiente de la carga de trabajo:
  - ⇨ 600 μs para la serie aritmética.
  - ⇨ 2 ms para *bubblesort*.
- **Modelos de fallos**:
  - ⇨ Para la técnica de Órdenes del simulador:
    - ❑ Transitorios: *Stuck-at* (‘0’, ‘1’), *delay*, *bit-flip*, *pulse* e *indetermination*.
    - ❑ Permanentes: *Stuck-at* (‘0’, ‘1’), *delay*, *open-line* e *indetermination*.
  - ⇨ Para los Perturbadores:
    - ❑ Transitorios: *Stuck-at* (‘0’, ‘1’), *delay*, *bit-flip* e *indetermination*.
    - ❑ Permanentes: *Stuck-at* (‘0’, ‘1’), *delay*, *open-line, indetermination*, *bridging* y *stuck-open*.
  - ⇨ Para los Mutantes: Cambios sintácticos en el código VHDL (en particular, los ocho modelos definidos en [Armstrong *et al.* 1992], y descritos en el apartado 5.4.2).

  En todos los casos, el modelo de fallo se selecciona de manera uniforme.

  Aunque aquí se refiere a la lista completa de los modelos de fallos aplicados, no se han aplicado todos los modelos en todos los experimentos. En algunos grupos de experimentos se han utilizado conjuntos diferentes de modelos, bien para comparar la incidencia del conjunto de modelos de fallos empleado, o por la inclusión posterior de nuevos modelos.
- **Instante de inyección**: Se genera de manera aleatoria, siguiendo una función de distribución Uniforme entre 0.0 ns y T_W (con T_W= 264 μs, 2 ms).

- **Duración de los fallos**: Se han inyectado fallos permanentes, transitorios generados aleatoriamente con una distribución Uniforme en los rangos [0.01T-1.0T] y [0.1T-10.0T], y transitorios de duración fija: 100.0T, donde T es el período de la señal de reloj.

7.4.1.3 Parámetros de análisis

Coinciden con los expuestos en el apartado 7.3.2.3

7.4.1.4 Resultados

En cada experimento se han calculado las mismas medidas de la Confiabilidad: el porcentaje de errores activados y sus latencias de propagación, las coberturas de detección y recuperación de errores (tanto de los mecanismos como globales), y las latencias de detección y recuperación de los errores. También se ha cumplimentado el grafo de predicados de los mecanismos de tolerancia a fallos.

Comparando los resultados obtenidos en los experimentos, se ha analizado la influencia en las medidas de la Confiabilidad de:

- La duración de los fallos.
- El conjunto de los modelos de fallos.
- La técnica de inyección.

A continuación se muestra una serie de tablas que muestran la influencia de la carga y la duración de los fallos en las medidas de la Confiabilidad. Se obtienen de unos experimentos donde se inyectan fallos transitorios (*stuck-at*, *bit-flip*, *indetermination* y *delay*) utilizando órdenes del simulador, donde se varían:

- Los rangos entre los que oscila la duración de los fallos:
  - ⇨ [0.01T-1.0T]
  - ⇨ [0.1T-10.0T]
- La carga de trabajo: Serie aritmética y *bubblesort*.

La Tabla 7.3 muestra los valores de las medidas de la Confiabilidad en función de la duración de los fallos. Se puede observar que:

- El porcentaje de errores activados (P_A) y las coberturas de los mecanismos $C_{d(mec)}$ y $C_{r(mec)}$ aumentan con la duración del fallo. En el caso de P_A es porque los fallos de menor duración tienen menor influencia en el funcionamiento del sistema. En las coberturas es porque los fallos de menor duración son más difíciles de detectar (y, por lo tanto, de recuperar). En cuanto a la
- Las coberturas globales $C_{d(sys)}$ y $C_{r(sys)}$ decrecen con la duración del fallo. Se debe a la disminución de los errores no efectivos, como se puede ver en los grafos que se muestran en la Figura 7.17, la Figura 7.18, la Figura 7.21 y la Figura 7.22.
- Las latencias no parecen mostrar ninguna dependencia de la duración del fallo, ya que existe la misma relación entre ellas: $l_r >> l_d > l_p$.

En la tabla no se aprecia ninguna dependencia con respecto a la carga. Se puede resaltar el hecho de que los valores obtenidos para *bubblesort* son superiores a los de la serie aritmética.

		Serie aritmética		Bubblesort	
		[0.01T-1.0T]	[0.1T-10.0T]	[0.01T-1.0T]	[0.1T-10.0T]
Medidas de la Confiabilidad	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 7.3: Medidas de la Confiabilidad en función de la carga de trabajo y la duración del fallo [Gil *et al.* 2000].

En la Tabla 7.4 se puede ver la contribución de cada uno de los mecanismos de detección de errores a la cobertura y la latencia (de detección). Es de destacar que la paridad es el mecanismo más efectivo, detectando más errores y con mucha menor latencia. Por otro lado, no parece que haya ninguna dependencia con la carga de trabajo.

Mecanismos de detección	Serie aritmética				Bubblesort			
	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]
Parity	61.35	56.21	3021	4077	62.60	53.69	7694	9687
Watchdog	38.65	43.79	76779	75888	37.40	46.31	77377	72756

Tabla 7.4: Contribución de los mecanismos de detección de fallos a la cobertura y las latencias [Gil *et al.* 2000].

Por su parte, la Tabla 7.5 muestra la contribución de los mecanismos de recuperación de errores en la cobertura y la latencia (de recuperación). Sobre todo para mayores duraciones de los fallos, el mecanismo más eficaz en cuanto al porcentaje de errores recuperados es el repuesto en frío (`Spare`), seguido del ciclo de recuperación y la paridad. La causa es el mayor índice de errores permanentes ocasionados por fallos de mayor duración. Como inconveniente, presenta una latencia muy superior al resto de los mecanismos.

Mecanismos de recuperación	Serie aritmética				Bubblesort			
	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]
Backoff	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
Spare	71.34	58.06	146667	13866	69.63	66.46	141346	133467

Tabla 7.5: Contribución de los mecanismos de recuperación de fallos a la cobertura y las latencias [Gil *et al.* 2000].

Las tablas que se exponen a continuación permiten ver la influencia de la duración de los fallos en las medidas de la Confiabilidad. En los experimentos de donde se obtienen se inyectan fallos permanentes (*stuck-at*, *indetermination*, *open-line* y *delay*) y transitorios (*stuck-at*, *bit-flip*, *indetermination* y *delay*) de duraciones diferentes, con una carga de la serie aritmética de seis enteros, y utilizando órdenes del simulador, donde se varía la duración de los fallos transitorios:

- En el rango [0.1T-10.0T]
- 100T

La Tabla 7.6 muestra los valores de las medidas de la Confiabilidad en función de la duración de los fallos. Se puede comprobar que los parámetros siguen la misma tendencia mostrada en la Tabla 7.3, a excepción de la cobertura de recuperación, que se reduce considerablemente en fallos permanentes. Este efecto puede ser causado por errores propagados a la CPU de repuesto, que impiden la recuperación del sistema.

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

Tabla 7.6: Medidas de la Confiabilidad en función de la duración del fallo [Baraza *et al.* 2000].

En la Tabla 7.7 se puede ver la contribución de cada uno de los mecanismos de detección de errores a la cobertura y la latencia (de detección). Se puede observar una dependencia de la duración del fallo en la efectividad de los mecanismos. Mientras en fallos (relativamente) "cortos" la paridad es el mecanismo más efectivo, en los fallos "largos" es el temporizador de guardia. En cualquier caso, las latencias de la paridad son muy inferiores a las del temporizador de guardia.

Mecanismos de detección	Errores detectados (%)			Latencia media (ns)		
	[0.1T-10.0T]	100T	Permanentes	[0.1T-10.0T]	100T	Permanentes
Parity	61.35	38.85	42.42	3021	6062	6210
Watchdog	38.65	61.15	57.58	76779	70165	66255

Tabla 7.7: Contribución de los mecanismos de detección de fallos a la cobertura y las latencias [Baraza *et al.* 2000].

La Tabla 7.8 muestra la contribución de los mecanismos de recuperación de errores en la cobertura y la latencia (de recuperación). El mecanismo más eficaz en cuanto al porcentaje de errores recuperados es el repuesto en frío (`Spare`), sobre todo con los fallos permanentes, ya que éstos ocasionan un mayor porcentaje de errores permanentes, que hacen activarse la CPU de repuesto. Como ya se vio en la Tabla 7.5, presenta el inconveniente de tener una latencia muy superior al resto de los mecanismos.

Mecanismos de recuperación	Errores recuperados (%)			Latencia media (ns)		
	[0.1T-10.0T]	100T	Permanentes	[0.1T-10.0T]	100T	Permanentes
Backoff	22.81	4.26	6.39	13916	81333	54810
Checkpoint	5.85	44.32	1.83	35932	46577	25194
Spare	71.34	51.42	91.78	146667	151244	128022

Tabla 7.8: Contribución de los mecanismos de recuperación de fallos a la cobertura y las latencias [Baraza *et al.* 2000].

El siguiente grupo de figuras muestra la influencia de la técnica de inyección, de la carga de trabajo y la persistencia de los fallos en las medidas de la Confiabilidad. Se corresponden con experimentos en los que se varían:

- La técnica de inyección:
  - ⇨ Órdenes del simulador.
  - ⇨ Perturbadores.
  - ⇨ Mutantes.
- La duración de los fallos:
  - ⇨ Transitorios, generados uniformemente en el rango [0.1T-10.0T].
  - ⇨ Permanentes.
- La carga de trabajo: Serie aritmética y *bubblesort*.

Los modelos de fallos utilizados en este conjunto de experimentos son:

- Órdenes del simulador:
  - ⇨ Transitorios: *delay*, *bit-flip*, *pulse* e *indetermination*.
  - ⇨ Permanentes: *stuck-at*, *delay*, *open-line* e *indetermination*.
- Perturbadores:
  - ⇨ Transitorios: *delay*, *bit-flip*, *pulse* e *indetermination*.
  - ⇨ Permanentes: *stuck-at*, *delay*, *open-line*, *indetermination*, *bridging* y *stuck-open*.
- Mutantes: cambios sintácticos en el código VHDL.

La Figura 7.11 muestra el porcentaje de errores activados en función de la técnica de inyección, la carga de trabajo y la duración de los fallos. Como se puede comprobar, P_A aumenta considerablemente con perturbadores y mutantes. Es decir, los fallos inyectados con estas técnicas tienen mayor influencia en el sistema que los inyectados con órdenes del simulador.

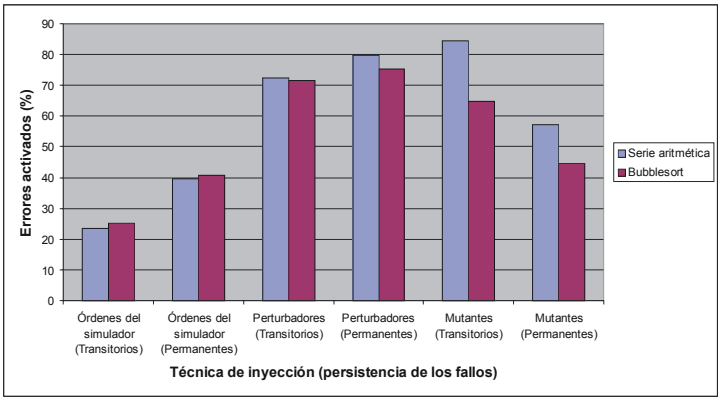

Figura 7.11: Porcentaje de errores activados en función de la técnica de inyección, la duración del fallo y la carga de trabajo [Gil *et al.* 2003a].

En la Figura 7.12 y en la Figura 7.13 se pueden ver respectivamente las coberturas de detección y recuperación en función de la técnica de inyección, la carga de trabajo y la duración de los fallos.

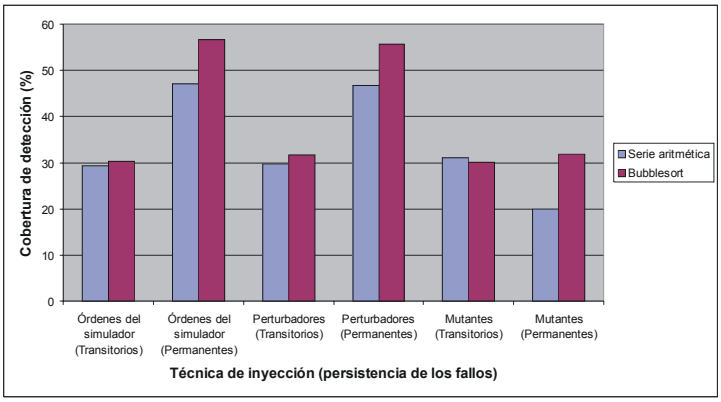

Figura 7.12: Cobertura de detección de errores en función de la técnica de inyección, la duración del fallo y la carga de trabajo [Gracia *et al.* 2001a].

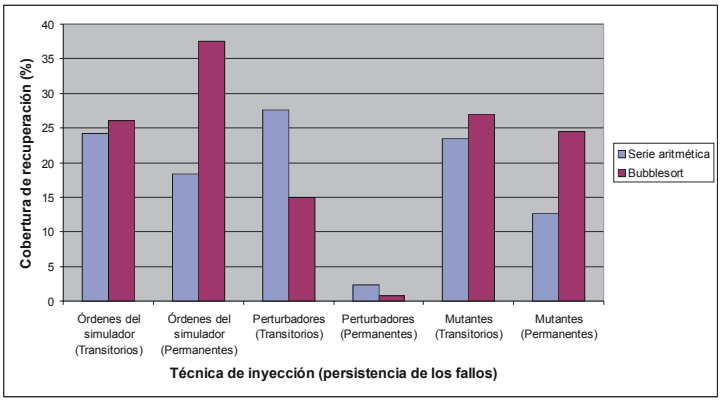

Figura 7.13: Cobertura de recuperación de errores en función de la técnica de inyección, la duración del fallo y la carga de trabajo [Gracia *et al.* 2001a].

Dos detalles son especialmente destacables:

- Tanto para C_d como para C_r existe una discrepancia en la inyección de fallos permanentes utilizando mutantes. Tal vez sería interesante analizar en detalle la representatividad de los modelos de fallos permanentes para mutantes.
- C_r presenta un fuerte descenso en los perturbadores permanentes (véase la Figura 7.13). La causa es que los fallos afectan al sistema de repuesto, aún cuando no se hayan inyectado fallos directamente sobre él.

Por su parte, de la Figura 7.14 a la Figura 7.16 se representan las latencias medias de propagación, detección y recuperación en función de la técnica de inyección, la carga de trabajo y la duración de los fallos.

De la Figura 7.14 se deduce que la latencia de propagación es mayor en órdenes del simulador que en las otras dos técnicas, excepto en algún caso como los perturbadores permanentes cuando la carga es *bubblesort*. La causa puede estar relacionada con los lugares de inyección.

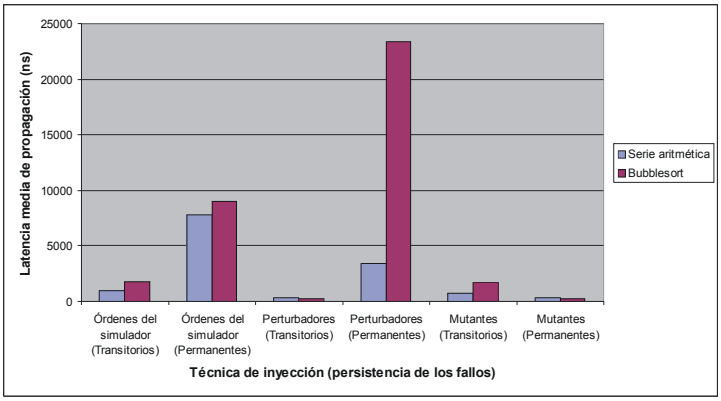

Figura 7.14: Latencia media de propagación de errores en función de la técnica de inyección, la duración del fallo y la carga de trabajo [Gil *et al.* 2003a].

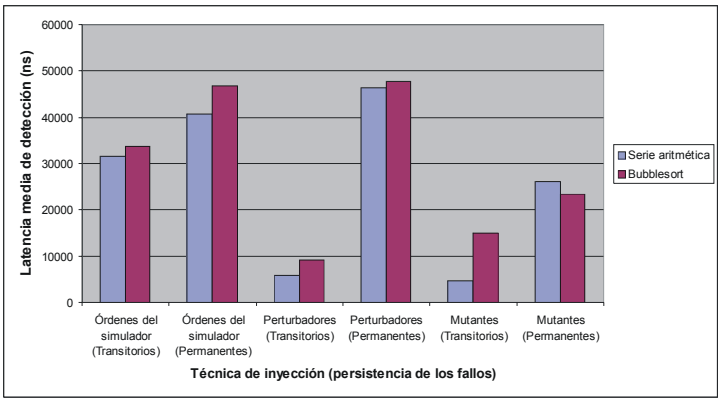

Figura 7.15: Latencia media de detección de errores en función de la técnica de inyección, la duración del fallo y la carga de trabajo [Gracia *et al.* 2001b].

De la Figura 7.15 y la Figura 7.16 se ve que, para fallos transitorios, las mayores latencias se observan con las órdenes del simulador, y las menores con los perturbadores. En cambio, para fallos permanentes no se distingue una dependencia clara de la técnica de inyección.

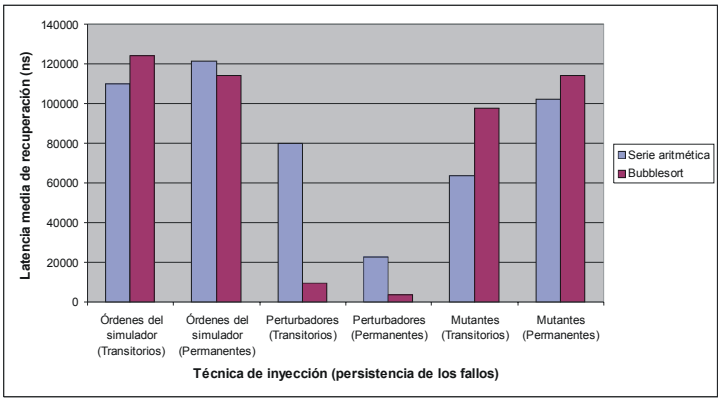

Figura 7.16: Latencia media de recuperación de errores en función de la técnica de inyección, la duración del fallo y la carga de trabajo [Gracia *et al.* 2001b].

De la Figura 7.17 a la Figura 7.24 se muestran algunos grafos de predicados de los mecanismos de tolerancia a fallos.

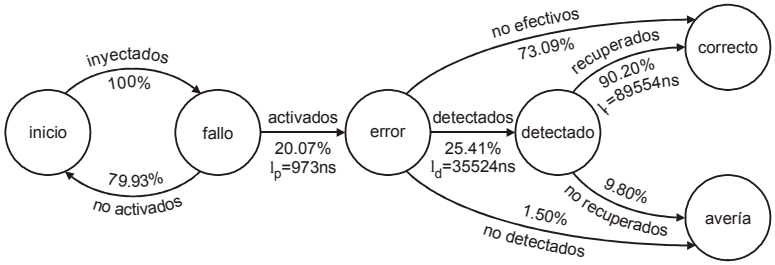

Figura 7.17: Grafo de predicados de los mecanismos de tolerancia a fallos. Técnica: Órdenes del simulador. Carga de trabajo: Serie aritmética. Duración de los fallos: Transitorios en el rango [0.01T-1.0T] [Gil *et al.* 2000].

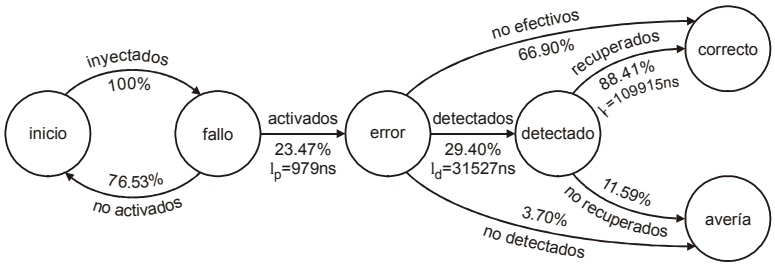

Figura 7.18: Grafo de predicados de los mecanismos de tolerancia a fallos. Técnica: Órdenes del simulador. Carga de trabajo: Serie aritmética. Duración de los fallos: Transitorios en el rango [0.1T-10.0T] [Gil *et al.* 2000].

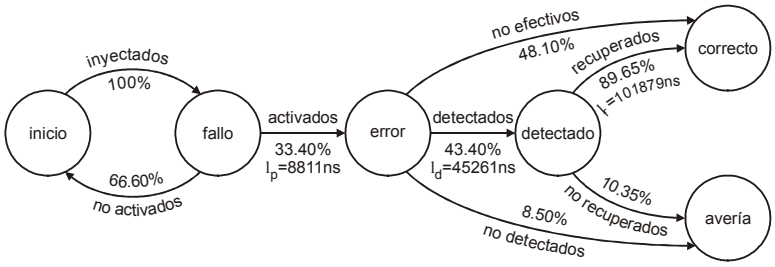

Figura 7.19: Grafo de predicados de los mecanismos de tolerancia a fallos. Técnica: Órdenes del simulador. Carga de trabajo: Serie aritmética. Duración de los fallos: Transitorios de duración 100T [Baraza *et al.* 2002].

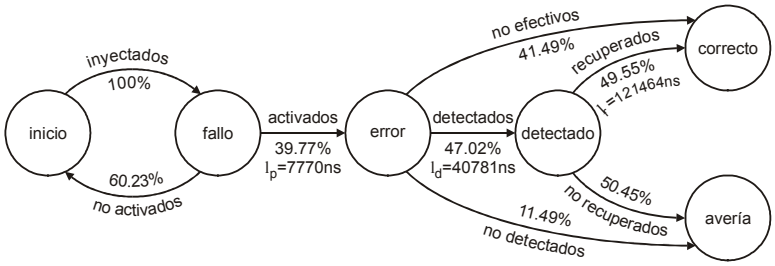

Figura 7.20: Grafo de predicados de los mecanismos de tolerancia a fallos. Técnica: Órdenes del simulador. Carga de trabajo: Serie aritmética. Duración de los fallos: Permanentes [Baraza *et al.* 2002].

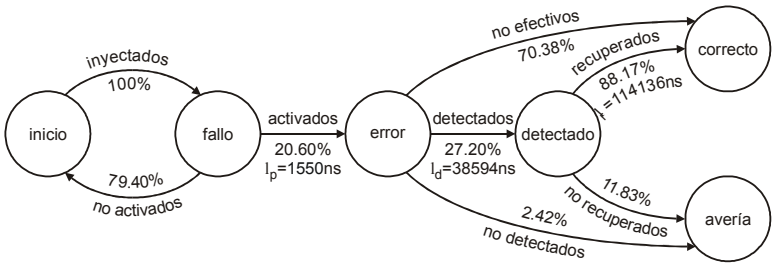

Figura 7.21: Grafo de predicados de los mecanismos de tolerancia a fallos. Técnica: Órdenes del simulador. Carga de trabajo: *Bubblesort*. Duración de los fallos: Transitorios en el rango [0.01T-1.0T] [Gil *et al.* 2000].

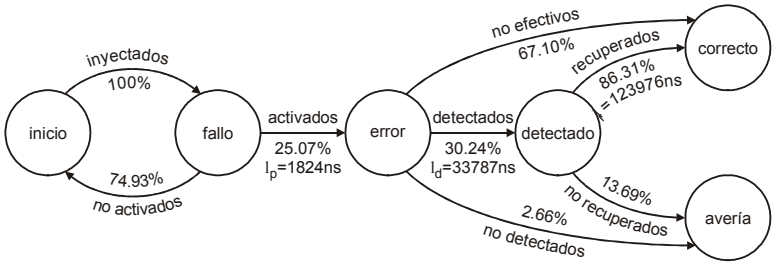

Figura 7.22: Grafo de predicados de los mecanismos de tolerancia a fallos. Técnica: Órdenes del simulador. Carga de trabajo: *Bubblesort*. Duración de los fallos: Transitorios en el rango [0.1T-10.0T] [Gil *et al.* 2000].

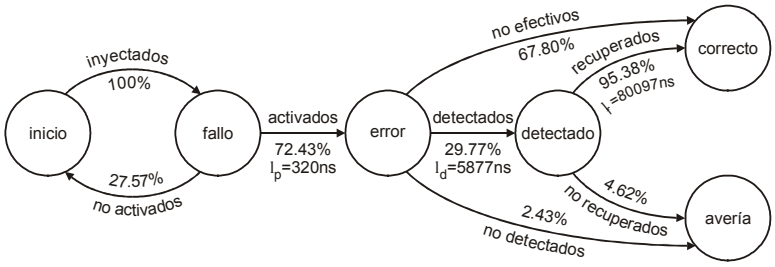

Figura 7.23: Grafo de predicados de los mecanismos de tolerancia a fallos. Técnica: Perturbadores. Carga de trabajo: Serie aritmética. Duración de los fallos: Transitorios en el rango [0.1T-10.0T] [Gil *et al.* 2003a].

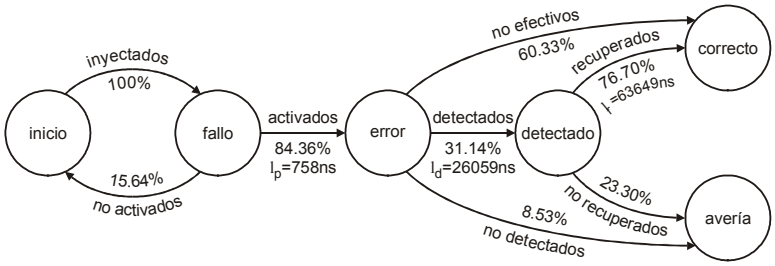

Figura 7.24: Grafo de predicados de los mecanismos de tolerancia a fallos. Técnica: Mutantes. Carga de trabajo: Serie aritmética. Duración de los fallos: Transitorios en el rango [0.1T-10.0T] [Gil *et al.* 2003a].

En particular, los grafos de la Figura 7.17, la Figura 7.18, la Figura 7.21 y la Figura 7.22 se corresponden con la inyección de fallos transitorios con órdenes del simulador, empleando dos cargas de prueba (la suma de la serie aritmética de seis enteros y el algoritmo de ordenación *bubblesort*) y con dos rangos de duración de los fallos ([0.01T-1.0T] y [0.1T-10.0T]).

Los grafos de la Figura 7.18 a la Figura 7.20 se refieren a la inyección de fallos de diferente duración (transitorios en el rango [0.1T-10.0T], transitorios de 100T y permanentes). En todos los experimentos se ha utilizado la suma de la serie aritmética de seis enteros y la técnica de órdenes del simulador.

Los grafos de la Figura 7.18, la Figura 7.23 y la Figura 7.24 pertenecen a la inyección de fallos transitorios de duración en el rango [0.1T-10.0T], con la serie aritmética de seis enteros como carga de trabajo, para las tres técnicas de inyección. Como se puede constatar, con las tres técnicas se producen un número no insignificante de errores no recuperados (e incluso no detectados). La causa principal es que el modelo bajo estudio es académico, y los mecanismos de detección de fallos que incorpora (detección de paridad) son insuficientes. Por tanto, es necesario añadir algún mecanismo interno de detección adicional, como por ejemplo excepciones.

Por último, en la Tabla 7.9 se muestra una comparación del coste temporal de las fases de simulación y análisis en diferentes experimentos, en función de la técnica de inyección y la carga de trabajo. Los experimentos se han llevado a cabo sobre un PC con procesador PII a 350 MHz y con 192 MB de memoria RAM, y en todos se han inyectado fallos transitorios en el rango [0.1T-10.0T].

	Fase de simulación		Fase de análisis	
Técnica de inyección	**Serie aritmética**	**Bubblesort**	**Serie aritmética**	**Bubblesort**
Órdenes 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 7.9: Comparación del coste temporal de un experimento de inyección en función de la técnica de inyección y la carga de trabajo. Duración de los fallos: Transitorios en el rango [0.1T-10.0T] [Gracia *et al.* 2001b].

7.4.2 Experimentos realizados sobre el microcontrolador de comunicaciones TTP™/C-C1

7.4.2.1 Objetivos

El objetivo principal del proyecto europeo FIT (*Fault Injection for TTA*, IST-1999-10748, 01/05/1999-30/04/2002), en el que ha participado el Grupo de Sistemas Tolerantes a Fallos (GSTF) ha sido la validación del controlador de comunicaciones TTP™/C desarrollado por TTTech [TTP/C C1 2002, TTP/C-C1 2001]. Para llevar a cabo la validación se han utilizado diversas técnicas de inyección entre las que se incluye la simulación de modelos en VHDL, en este caso mediante la herramienta VFIT.

Los resultados que se presentan en este apartado son fruto de los experimentos realizados para dicha validación. Algunos pertenecen a informes del proyecto [FIT 2002a, FIT 2002b] y otros a publicaciones en congresos [Gracia *et al.* 2002c, Gracia *et al.* 2003].

Los experimentos de inyección debían llevarse a cabo sobre dos versiones del mencionado prototipo: TTP/C-C1 y TTP/C-C2. Sobre el primer prototipo se realizaron diversas campañas que arrojaron resultados muy satisfactorios, tanto desde el punto de vista de la técnica de inyección como de la propia herramienta. Además, se detectaron errores de diseño que (se supone) fueron subsanados en la segunda versión. Sin embargo, por diversos problemas técnicos no fue posible disponer de un *cluster* (véase el apartado 7.2.5) con el modelo del TTP/C-C2 similar en configuración y carga al del TTP/C-C1. Por ello, no se ha incluido la descripción del segundo prototipo en el apartado 7.2, ni se especificarán los resultados obtenidos en detalle. En cambio, sí se hará alguna alusión a algún resultado especialmente importante.

Dada la excesiva complejidad del modelo[86], su validación de manera global se presentó inabordable, por lo que se realizaron una serie de experimentos preliminares para localizar los componentes del modelo más sensibles a los fallos transitorios. Se planteó entonces la validación del modelo por bloques, realizando campañas específicas sobre cada uno de los componentes críticos escogidos[87]. Los componentes que demostraron mayor sensibilidad a los fallos transitorios fueron las unidades PCU (*Protocol Control Unit*), y en particular, su registro de

[86] Este problema fue común a los modelos de las dos versiones del microcontrolador, más aún si cabe al del TTP/C-C2.

[87] Obsérvese la diferencia de objetivo entre la validación del sistema completo (tal como se hizo para el modelo del MARK2 tolerante a fallos), y la validación parcial (sólo de algunos componentes individuales) realizada sobre el controlador TTP/C. En el primer caso se lleva a cabo una *Predicción de fallos*, mientras que en el segundo la técnica de validación utilizada es la *Eliminación de fallos* (véase el capítulo 2).

instrucciones (que en adelante se denominará IR PCU), CRC, TCU (*Time Control Unit*) y el banco de registros general, o Register File (de aquí en adelante RF). Para probar la eficacia del guardian de bus (BG, o *Bus Guardian*), los experimentos sobre el CRC se realizaron con y sin el BG activado.

7.4.2.2 Parámetros de inyección

En el caso del controlador TTP/C-C1, los parámetros comunes (el período de la señal de reloj, la carga de trabajo y su duración, el tiempo de simulación) no son relevantes, por lo que no se especificarán. Sólo merece la pena destacar dos detalles:

- Únicamente se ha utilizado órdenes del simulador como técnica de inyección.
- La duración de los fallos ha sido generada aleatoriamente utilizando una distribución Uniforme en el rango [1/2 SRU *slot* - 1 SRU *slot*][88].

Los parámetros que han variado en alguno de los experimentos son el número de fallos inyectados, el lugar de inyección (todas las señales y variables de un determinado componente), los modelos de fallo y el instante de inyección. La Tabla 7.10 muestra los parámetros correspondientes a cada experimento.

	Experimento(*)					
	PCU	**IR PCU**	**TCU**	**CRC (BG)**	**CRC (sin BG)**	**RF**
Nº de fallos	3000	3000	2000	2000	1000	2000
Modelos de fallos	Bit-flip, Pulse, Indetermination, Delay	Bit-flip, Pulse, Indetermination, Delay	Bit-flip, Pulse, Indetermination, Delay	Bit-flip, Pulse, Indetermination, Delay	Bit-flip, Pulse, Indetermination, Delay	Bit-flip, Indetermination, Delay
Instante de inyección	Aleatorio, generado uniformemente en la 1ª ronda de TDMA	Aleatorio, generado uniformemente en la 1ª ronda de TDMA	Aleatorio, generado uniformemente en la 1ª ronda de TDMA	Aleatorio, generado uniformemente en la 1ª ronda de TDMA	Aleatorio, generado uniformemente en la 1ª ronda de TDMA	Aleatorio, generado uniformemente en la 2ª ronda de TDMA

(*) El nombre de cada experimento está determinado por el del componente donde se inyectan los fallos.

Tabla 7.10: Parámetros de inyección de los experimentos realizados sobre el controlador de comunicaciones TTP/C-C1.

7.4.2.3 Parámetros de análisis

El dato más interesante que vale la pena reseñar de estos parámetros es la especificación de los mecanismos de detección de errores (EDM) del sistema: HE, PE, BE, ML, Combination y Fail Silent. Obsérvese que se han añadido dos nombres a los referidos en el apartado 7.2.5 (Combination y Fail Silent). Estos no representan realmente a ningún mecanismo, sino casos especiales. En particular, Combination indica la activación de más de un EDM, y Fail Silent significa que el sistema ha quedado en silencio, a pesar de no haberse activado ningún EDM.

El resto de los parámetros de análisis son demasiado dependientes del sistema y no aportan demasiada luz acerca del análisis.

[88] La duración de 1 SRU *slot* es de aproximadamente 165 μs.

7.4.2.4 Resultados

El objetivo principal de los experimentos no es el cálculo de las coberturas ni de las latencias, sino detectar posibles defectos de tolerancia, especialmente violaciones del silencio ante avería (*fail silence*). Es decir, hay que descubrir si la aparición de fallos puede ocasionar averías, y detectar las causas: defectos de diseño, de implementación, etc.

Para ello, los datos que conviene observar son los porcentajes que aparecen en el grafo de predicados de los mecanismos de tolerancia a fallos: los errores activados (etiquetados como `activados`), los errores activados no efectivos (`no efectivos`), los detectados (`detectados`), los no detectados (`no detectados`), los detectados pero no recuperados (etiqueta `no recuperados`) y los detectados no efectivos. En principio éstos últimos se corresponden con los etiquetados como `recuperados`. Lo que ocurre es que como el controlador no dispone de mecanismos de recuperación de errores (véase el apartado 7.2.5), los errores que se recuperan son debidos a la redundancia intrínseca del sistema, no provocando una avería.

Como se puede ver en la Tabla 7.11, salvo en `RF`, en todos los demás componentes se han producido averías. Dichas averías se manifiestan de dos maneras:

- **Violación del silencio ante averías** en el dominio del tiempo. Se produce un colapso de todos los nodos del *cluster*, que quedan en silencio. Se presenta de dos formas, durando una o dos rondas de TDMA. La causa del colapso es una incorrecta implementación del denominado *clique avoidance algorithm*, que es un algoritmo encargado del control de la comunicación.
- **SoS** (*Slightly off Specification*). Se produce cuando el guardian de bus del nodo afectado por el fallo le autoriza a transmitir en un *slot* que no es el suyo. Esta avería puede ocasionar colisiones de tramas.

	Experimento					
	PCU	**IR PCU**	**TCU**	**CRC (con BG)**	**CRC (sin BG)**	**RF**
Errores Activados	38.17	38.10	39.15	37.15	46.40	28.75
Errores No efectivos	6.89	14.00	10.35	22.88	16.59	0.00
Errores Detectados	93.02	85.13	89.14	75.10	82.11	100.00
Errores No detectados	**0.09**	**0.87**	**0.51**	**2.02**	**1.30**	**0.00**
Errores No recuperados	**1.13**	**1.75**	**0.86**	**0.72**	**0.26**	**0.00**
Errores Detectados no efectivos	98.87	98.25	99.14	99.28	99.74	100.00

Tabla 7.11: Resultados de los seis experimentos realizados sobre el controlador de comunicaciones TTP/C-C1 [FIT 2002a, FIT 2002b].

Puesto que el número total de averías (y su gravedad) no es insignificante, es preciso resolver el problema. De hecho, se supone que en la segunda versión del controlador (el TTP/C-C2) se han resuelto éstos entre otros problemas de la primera versión. Para verificarlo, se han

realizado experimentos de inyección en el TTP/C-C2 para comprobarlo. En estos experimentos no se han detectado violaciones del silencio ante averías, pero sí averías SoS [FIT 2002b].

Atendiendo a la sensibilidad de los diferentes componentes sobre los que se han inyectado fallos, el IR PCU y el CRC (sobre todo con el BG activado) son los más sensibles, ya que presentan un mayor índice de averías.

A continuación se muestran una serie de figuras relacionadas con cuatro de los experimentos: PCU, IR PCU, TCU y CRC (con BG).

Los grafos de predicados de los mecanismos de tolerancia a fallos representados de la Figura 7.25 a la Figura 7.28 permiten comparar las latencias de propagación y detección de errores obtenidas en los cuatro experimentos mencionados. Obsérvese que no hay latencias de recuperación. Se debe a que los mecanismos de recuperación están implementados en niveles superiores (en el *host controller*).

De los grafos se puede destacar que:

- El porcentaje de errores activados es muy similar en todos los experimentos.
- El porcentaje de errores detectados, en cambio, es muy dispar. De los cuatro módulos, CRC e IR PCU son los más sensibles a los fallos, ya que presentan un valor sensiblemente menor. Esta observación se confirma con el hecho de que el porcentaje de averías (generada mediante la suma de los porcentajes no detectado y no recuperado) es superior al de los otros módulos. De aquí se desprende que estos dos componentes son los más críticos.

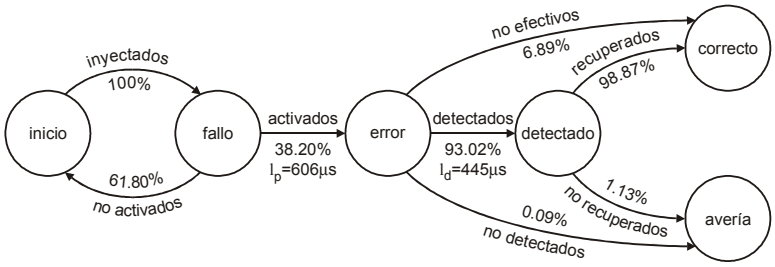

Figura 7.25: Grafo de predicados de los mecanismos de tolerancia a fallos. Experimento: PCU [Gracia *et al.* 2002c].

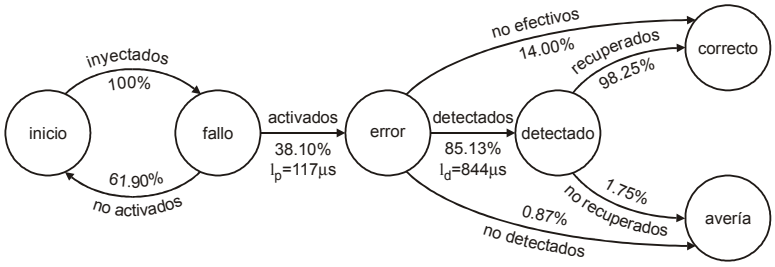

Figura 7.26: Grafo de predicados de los mecanismos de tolerancia a fallos. Experimento: IR PCU [Gracia *et al.* 2002c].

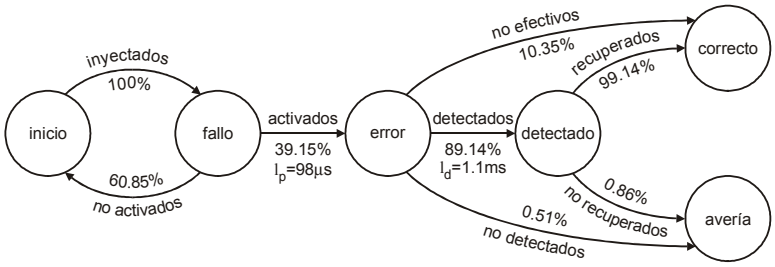

Figura 7.27: Grafo de predicados de los mecanismos de tolerancia a fallos. Experimento: TCU [Gracia *et al.* 2003].

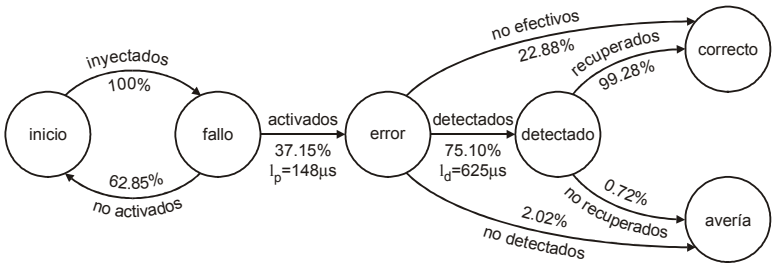

Figura 7.28: Grafo de predicados de los mecanismos de tolerancia a fallos. Experimento: CRC (con BG) [Gracia *et al.* 2003].

- Las latencias tampoco presentan un comportamiento homogéneo. Mientras los fallos se propagan en la TCU más rápidamente que en el resto (siendo los fallos en la PCU los más lentos en propagarse), con la latencia de detección ocurre el fenómeno opuesto. Los errores en la PCU son los más rápidos en ser detectados, y en la TCU los más lentos.
- En todos los experimentos se da un elevadísimo porcentaje de errores recuperados. Si se tiene en cuenta que el controlador no dispone de mecanismos de recuperación, hay que deducir que se corrigen por la redundancia intrínseca del diseño.

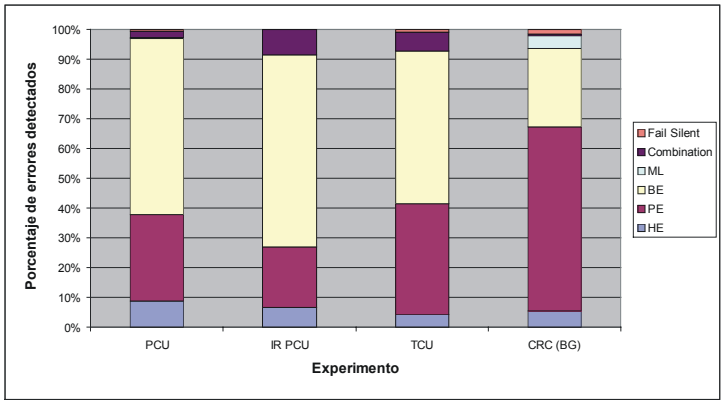

Figura 7.29: Porcentajes de errores detectados por los diferentes mecanismos de detección de errores (EDM) [Gracia *et al.* 2002c, Gracia *et al.* 2003].

Por último, en la Figura 7.29 se puede ver la distribución de los fallos detectados por los EDM (mecanismos de detección de errores). Como se puede comprobar en la figura, en todos los experimentos excepto en el CRC se observa un comportamiento homogéneo. Esta tenden-

cia común consiste en la predominancia de los mecanismos BE, seguida por PE. En el CRC, en cambio, se invierten los papeles.

7.5 Análisis de la representatividad de los modelos de fallos a nivel RT

Los experimentos que se presentan en este bloque pertenecen a una de las últimas líneas de investigación abiertas por el Grupo de Sistemas Tolerantes a Fallos. De hecho, algunos de ellos son recientes y todavía no se han publicado.

La mencionada línea de investigación está enclavada en el proyecto europeo DBench (*Dependability Benchmarking*, IST-2000-25425, 01/01/2001-31/12/2003 [DBench 2003]). Con ella se pretende (partiendo de la inyección de fallos en modelos de sistemas a nivel de puerta) analizar la evolución de los fallos inyectados hacia el nivel RT (y superiores), para establecer los modelos de fallos más representativos en dichos niveles.

Para realizar las inyecciones se han escogido una serie de modelos estructurales (y sintetizables) de algunos microcontroladores y microprocesadores. A continuación se describe la forma de operar sobre cada uno de estos modelos.

En una primera fase, se trata de inyectar fallos sobre la lógica combinacional del modelo, y estudiar su propagación a los registros de la CPU. Desde el punto de vista de la herramienta, se trata de un análisis similar al del síndrome de errores, comentado anteriormente, aunque en esta ocasión el objetivo es estudiar la representatividad de los fallos.

En una segunda fase, todavía en curso, se pretende inyectar fallos en los registros de la CPU (distinguiendo entre los que son accesibles por programación y los que quedan "ocultos"), observando su incidencia en las averías ocasionadas. Esto puede ser importante para detectar deficiencias en algunas técnicas de inyección ampliamente utilizadas, como SWIFI (*software implemented fault injection*) o SCIFI (*scan-chain implemented fault injection*).

Los experimentos mostrados en este apartado se corresponden con algunos de los efectuados sobre los microcontroladores PIC16C5X y MC8051, en particular de la primera fase.

Tras la finalización de los experimentos pertenecientes a la segunda fase, se llevará a cabo una nueva tanda de experimentos sobre otros microprocesadores, como LEON-2 [LEON-2 2003] y HORUS, un microprocesador RISC basado en el MIPS R3000 [MIPS 2001] desarrollado en el GSTF [Rodríguez *et al.* 2002].

7.5.1 Experimentos realizados sobre el microcontrolador PIC16C5X

7.5.1.1 Objetivos

Los objetivos que se pretende alcanzar con estos experimentos coinciden con los de la línea de investigación, explicados en el apartado 7.5. Algunos de los resultados obtenidos se han publicado, bien en informes del proyecto DBench [DBench 2002], o en congresos [Gracia *et al.* 2002a, Gracia *et al.* 2002b].

7.5.1.2 Parámetros de inyección

Los parámetros comunes a todos los experimentos son:

- **Técnica de inyección aplicada**: Órdenes del simulador.
- **Número de fallos inyectados**: 3000.

Los parámetros que han variado en alguno de los experimentos son:

- **Período de la señal de reloj (T)**:
  - ⇨ 200 ns (f = 5 MHz).
  - ⇨ 100 ns (f = 10 MHz). En los experimentos donde no se indique explicitimente la frecuencia de funcionamiento, éste será el período de la señal de reloj.
  - ⇨ 50 ns (f = 20 MHz).
- **Carga de trabajo**:
  - ⇨ Cálculo de la serie aritmética de *n* enteros, con *n* = 10.
  - ⇨ Algoritmo de ordenación de la burbuja (*bubblesort*) para *n* enteros, con *n* = 10.
- **Duración de la carga de trabajo (T_W)**:
  - ⇨ 5.2 μs para la serie aritmética.
  - ⇨ 80 μs para *bubblesort*.
- **Duración de la simulación (T_S)**:
  - ⇨ 6 μs para la serie aritmética.
  - ⇨ 82 μs para *bubblesort*.
- **Lugares de inyección**: Todas las señales combinacionales de la ALU y en la señal de reloj.
- **Modelos de fallos**:
  - ❑ Transitorios: *Delay*, *pulse* e *indetermination*.
  - ❑ Permanentes: *Stuck-at* (‘0’, ‘1’), *delay*, *open-line* e *indetermination*.
- **Instante de inyección**: Se genera de manera aleatoria, siguiendo una función de distribución Uniforme entre 0.0 ns y T_W (con T_W=5.2 μs, 80 μs).
- **Duración de los fallos**: Se han inyectado fallos con tres tipos de duraciones:
  - ⇨ Permanentes.
  - ⇨ Transitorios de duración variable, generada aleatoriamente con una distribución Uniforme en los rangos [0.1T-1.0T], [1.0T-10.0T], [10.0T-20.0T], donde T es el período de la señal de reloj.
  - ⇨ Transitorios de duración fija, con los valores 10 ns, 100 ns y 1 μs (1000 ns).

7.5.1.3 Parámetros de análisis

En todos los experimentos se han considerado los mismos parámetros:

- **Lugares donde se detectan los errores**: Los registros del microcontrolador.
- **Clasificación de los fallos**: Los lugares de inyección se han clasificado en:
  - ⇨ ALU: Las señales de la ALU.
  - ⇨ CLK: La señal de reloj.
- **Clasificación de los errores**: Se consideran cuatro clases de errores propagados a los registros:
  - ⇨ Bit-flip.
  - ⇨ Indetermination.
  - ⇨ Delay.
  - ⇨ Open-line.

7.5.1.4 Resultados

VFIT genera en cada experimento los siguientes datos:

- El número de registros alterados (N_a). Un registro se considera alterado si tiene un fallo en al menos un bit.
- El número de fallos propagados (N_p). Se considera que un fallo se ha propagado si tras la inyección se altera al menos un registro.
- El número de averías ocasionadas por los fallos propagados (N_{ap}). Se considera que se ha producido una avería cuando el resultado de la ejecución de la carga de trabajo es erróneo.
- El porcentaje de fallos propagados (P_p), determinado por la expresión:

$$P_p = \frac{N_p}{N} \times 100 \tag{7.1}$$

  donde N es el número de fallos inyectados; en este caso N = 3000.

- La multiplicidad (M), o número promedio de registros alterados por cada fallo propagado. Está definido por la expresión:

$$M = \frac{N_a}{N_p} \tag{7.2}$$

- El porcentaje de averías provocadas por los fallos propagados. Está definido por la expresión:

$$P_{ap} = \frac{N_{ap}}{N} \times 100 \tag{7.3}$$

Con los datos obtenidos en el conjunto de los experimentos se han llevado a cabo diferentes estudios.

Por un lado, se ha analizado la influencia de la carga de trabajo, la frecuencia del reloj, la duración de los fallos o el modelo de fallo inyectado en los parámetros arriba indicados (N_p, N_a, N_{ap}, P_p, M y P_{ap}). Las gráficas de la Figura 7.30 a la Figura 7.33 muestran algunos de estos resultados.

Como se puede apreciar en la Figura 7.30, el porcentaje de errores propagados aumenta con la duración de los fallos. Asimismo, se observan diferencias no despreciables en los resultados para ambas cargas, especialmente en los fallos permanentes.

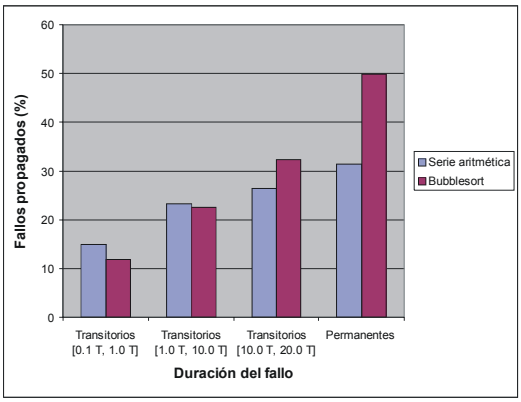

Figura 7.30: Influencia de la duración de los fallos y la carga en el porcentaje de fallos propagados.

En la Figura 7.31 se observa una dependencia de la multiplicidad con la duración del fallo, si bien en este caso es inversa (la multiplicidad disminuye con la duración de los fallos). La causa es que, cuanto mayor es la duración de los fallos, el número de errores propagados aumenta en mayor medida que el de registros afectados, por lo que la relación N_a/N_p disminuye. En cuanto a la dependencia de la carga, las diferencias son mímimas, si bien los valores para *bubblesort* son siempre ligeramente mayores que para la serie aritmética. En cualquier caso, se dan valores comprendidos entre 2 y 6 registros, lo que da idea de la presencia de fallos múltiples en los registros.

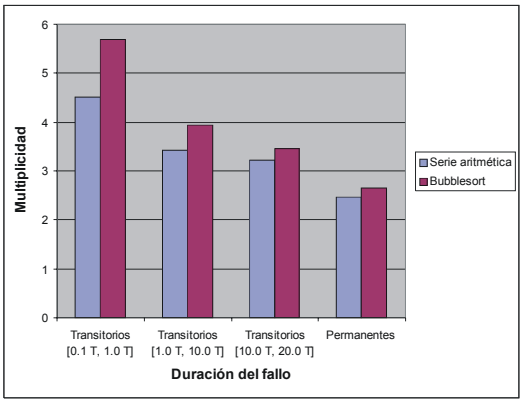

Figura 7.31: Influencia de la duración de los fallos y la carga en la multiplicidad.

La Figura 7.32 y la Figura 7.33 muestran la influencia de la frecuencia de funcionamiento en dos de los parámetros ya analizados.

La Figura 7.32 representa cómo afecta la frecuencia de funcionamiento al porcentaje de errores propagados. Se puede apreciar que, mientras en los fallos transitorios se produce un aumento del porcentaje con la frecuencia, en los permanentes no hay ninguna variación. Además, como cabía esperar, el porcentaje de fallos propagados también se incrementa con la duración de los fallos. La influencia de la frecuencia observada concuerda con lo predicho en el capítulo 4. En efecto, el incremento de la frecuencia aumenta la probabilidad de captura de datos erróneos transitorios generados en la lógica combinacional. Por el contrario, la captura de fallos permanentes no depende de la frecuencia, ya que los fallos no desaparecen.

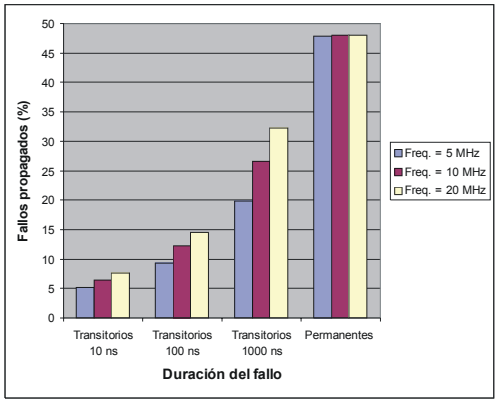

Figura 7.32: Influencia de la frecuencia de reloj y la duración de los fallos en el porcentaje de fallos propagados. Carga de trabajo: *Bubblesort*.

Por su parte, en la Figura 7.33 se puede ver su incidencia en el porcentaje de averías ocasionadas por los errores propagados. Se puede apreciar que el comportamiento es idéntico al observado en el caso anterior.

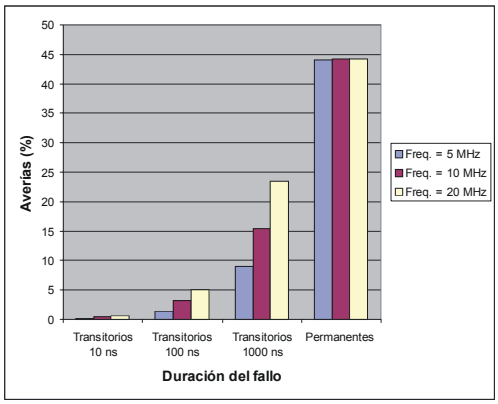

Figura 7.33: Influencia de la frecuencia de reloj y la duración de los fallos en el porcentaje de averías ocasionadas por los fallos propagados. Carga de trabajo: *Bubblesort*.

En cuanto al estudio de la representatividad de los modelos de fallos en el nivel RT, se han realizado diversos experimentos en los que se han variado la persistencia de los fallos y la carga de trabajo.

En la Figura 7.34 se puede observar cómo se manifiestan en el nivel RT los fallos inyectados en el nivel lógico, para todas las combinaciones de duración de fallo y carga de trabajo ejecutada.

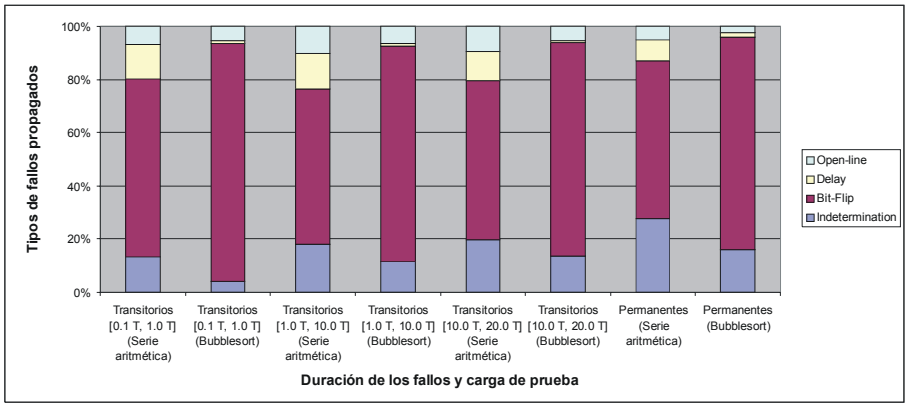

Figura 7.34: Distribución de los tipos de fallos propagados a los registros.

Los tipos de fallos propagados son cuatro: *bit-flip*, *indetermination*, *delay* y *open-line*, si bien es conveniente hacer algunas matizaciones:

- Las indeterminaciones permanentes en el modelo en VHDL se deberían traducir en *bit-flips* en los registros reales, ya que es prácticamente imposible que se alcance un estado metaestable permanente.
- En cambio, las indeterminaciones transitorias sí son posibles, debidas a violaciones de los márgenes de seguridad de los biestables, como se explicó en el capítulo 4 (más concretamente en el apartado 4.3.3).
- Los fallos de tipo *delay* en los registros están causados principalmente por la inyección de fallos *delay* en la señal de reloj.
- Los circuitos abiertos (fallos *open-line*) son ficticios, porque en la mayoría de los casos se deben a la pérdida de sincronización en la activación de las salidas triestado de los registros. El resto se manifiestan en realidad como *bit-flips* o como indeterminaciones transitorias

Cuando la carga de trabajo es *bubblesort*, los errores propagados son *bit-flip* e *indetermination* casi exclusivamente (la suma de sus porcentajes supone casi el 100%). Con la serie aritmética, estos mismos dos tipos son los mayoritarios, si bien la suma de los porcentajes de ambos modelos es aproximadamente del 80 %, lo que deja alrededor de un 20 % para los tipos *delay* y *open-line*.

En cambio, al comparar el efecto de la persistencia de los fallos no se detectan variaciones resaltables en los tipos de fallos propagados.

Con lo visto, se ha comprobado, pues, que la mayoría de los fallos se manifiestan en los registros como *bit-flips*, como era de esperar. No obstante, aunque en una menor proporción, también se manifiestan otros modelos (especialmente en fallos transitorios), como *indetermination* y *delay*. La duda surge a la hora de decidir en qué medida habría que tenerlos en cuenta en los modelos RT. El autor de la presente tesis opina que dependerá del sistema bajo estudio y del nivel de precisión que se quiere alcanzar. En cualquier caso, se trata de un tema que re-

quiere un estudio más profundo, basado en simulaciones electrónicas, como ya se comentó en el capítulo 4.

7.5.2 Experimentos realizados sobre el microcontrolador MC8051

7.5.2.1 Objetivos

Los objetivos buscados con los experimentos realizados sobre el microcontrolador MC8051 son los mismos que para el PIC16C5X. De hecho, los parámetros utilizados son los mismos (salvo aquéllos que dependen directamente del sistema), por lo que no se mencionarán aquí.

Como se mencionó en el apartado 7.5, todos los experimentos están enclavados en una línea de trabajo actualmente en desarrollo. De hecho, ningún resultado de los experimentos que se presentan a continuación ha sido todavía publicado.

7.5.2.2 Parámetros de inyección

Salvo pequeñas variaciones debidas a la diferencia de procesador, los parámetros de inyección (tanto los fijos como los variables) coinciden exactamente con los utilizados para el PIC16C5X. Las principales diferencias hacen referencia a:

- **Duración de la carga de trabajo (T_W)**:
  - ⇨ 15 µs.
  - ⇨ 130 µs.
- **Duración de la simulación (T_S)**:
  - ⇨ 17 µs.
  - ⇨ 132 µs.

7.5.2.3 Resultados

Para poder comparar los resultados, se han repetido los mismos experimentos que para el modelo del PIC16C5X, generando los mismos datos: número de registros alterados (N_a), número y porcentaje de fallos propagados (N_p y P_p), multiplicidad (M), número y porcentaje de averías ocasionadas por los fallos propagados (N_{ap} y P_{ap}), y los modelos de los fallos propagados a los registros. También se ha variado los mismos parámetros de la inyección (la carga de trabajo, la frecuencia del reloj y la duración de los fallos), para analizar su influencia en los datos obtenidos.

En cuanto a la influencia de la carga y la duración de los fallos en N_p, N_a, N_{ap}, P_p, M y P_{ap}, las gráficas de la Figura 7.35 a la Figura 7.38 (análogas a las mostradas de la Figura 7.30 a la Figura 7.33) permiten comparar el comportamientos de ambos microcontroladores.

Como sucedía en el PIC16C5X, el porcentaje de fallos propagados (en la Figura 7.35) aumenta con la duración del fallo. Sin embargo, en el MC8051 se observan mayores diferencias en la influencia de la carga, dándose valores particularmente altos para la suma de la serie aritmética.

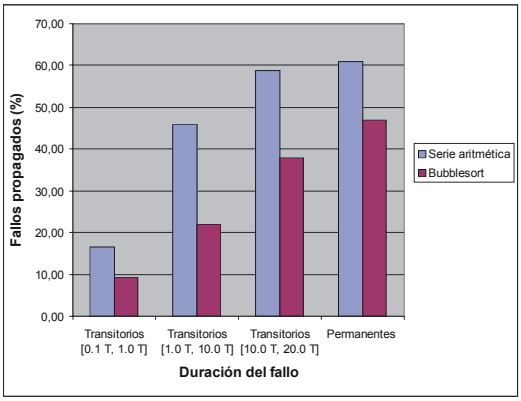

Figura 7.35: Influencia de la duración de los fallos y la carga en el porcentaje de fallos propagados.

La tendencia de los valores obtenidos para la multiplicidad (en la Figura 7.36) también coincide con los del PIC16C5X (en la Figura 7.31): la multiplicidad disminuye con la duración de los fallos, y los valores obtenidos para *bubblesort* son ligeramente mayores que los de la suma de la serie aritmética. Globalmente, se han observado valores comprendidos entre 3 y 8 registros.

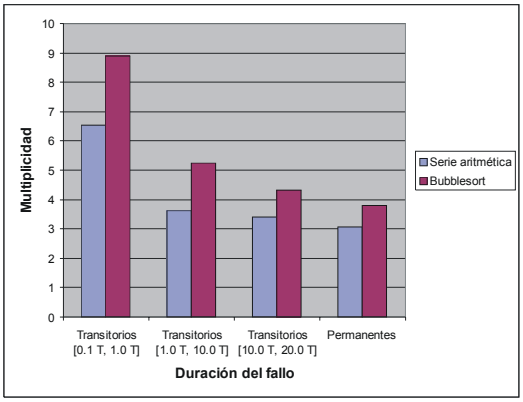

Figura 7.36: Influencia de la duración de los fallos y la carga en la multiplicidad.

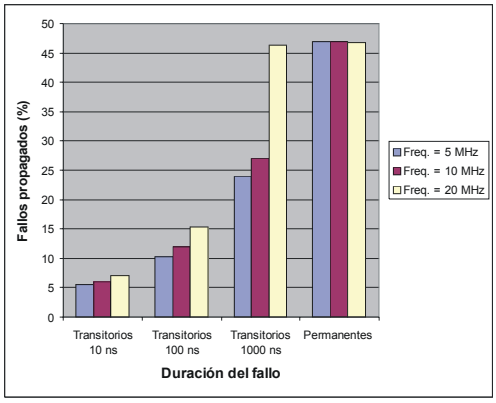

Figura 7.37: Influencia de la frecuencia de reloj y la duración de los fallos en el porcentaje de fallos propagados. Carga: *Bubblesort*.

La Figura 7.37 (análoga a la Figura 7.32 para el PIC16C5X) representa cómo afecta la frecuencia de funcionamiento al porcentaje de fallos propagados. Se puede apreciar, como en aquélla, que el porcentaje de fallos propagados aumenta con la frecuencia en los fallos transitorios, mientras que en los permanentes no tiene ninguna influencia.

También hay coincidencia en cómo influyen la frecuencia de funcionamiento y la duración de los fallos en el porcentaje de averías ocasionadas por los fallos propagados (en la Figura 7.38). Al igual que en el PIC16C5X (en la Figura 7.33), ésta aumenta tanto con la duración del fallo como con la frecuencia.

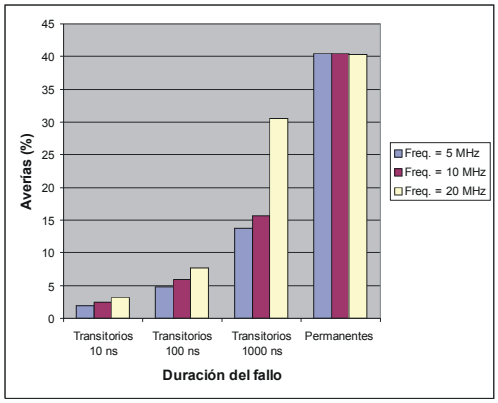

Figura 7.38: Influencia de la frecuencia de reloj y la duración de los fallos en el porcentaje de averías ocasionadas por los fallos propagados. Carga: *Bubblesort*.

En lo que se refiere al estudio de la representatividad de los modelos de fallos en el nivel RT, mostrado en la Figura 7.39, el comportamiento es muy similar al del PIC16C5X (en la Figura 7.34):

- Los modelos mayoritarios son *bit-flip* y, en menor proporción, *indetermination*.
- Los modelos *delay* y *open-line* tienen cierta relevancia en la suma de la serie aritmética, pero ninguna en *bubblesort*.
- No hay influencia significativa de la duración de los fallos.

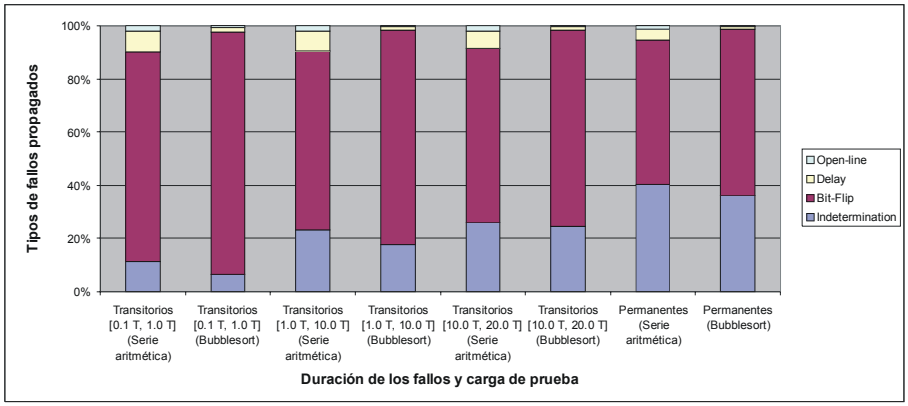

Figura 7.39: Distribución de los tipos de fallos propagados a los registros.

7.6 Resumen y conclusiones

En este capítulo se han expuesto algunos de los resultados más representativos obtenidos de la utilización de VFIT para inyectar fallos en diversos modelos en VHDL.

Es de destacar la variedad de los modelos de sistemas sobre los que se ha aplicado la herramienta. Se han inyectado fallos sobre modelos de microcontroladores comerciales (PIC y 8051), sistemas computador basados en microprocesadores académicos (MARK2) y un microcontrolador de comunicaciones (de la arquitectura TTA) TTP/C. Los modelos también se pueden clasificar siguiendo otros criterios, como el fin con el que fueron realizados (u objetivo) o su complejidad.

Atendiendo al objetivo de los modelos, se pueden distinguir modelos de simulación (los basados en el MARK2 y el del controlador TTP/C) y de síntesis (PIC y 8051). La complejidad de los mismos se puede clasificar en baja (los basados en el MARK2), media (PIC y 8051) y alta (controlador TTP/C).

Los resultados mostrados cubren las dos clases de análisis que se puede llevar a cabo con VFIT: estudio del síndrome de error (PIC y 8051) y validación de un sistema tolerante a fallos (el sistema tolerante a fallos basado en el MARK2 y el microcontrolador TTP/C).

Respecto a la validación del sistema computador basado en el microprocesador MARK2, se ha realizado una *Predicción de fallos*. Se han medido los porcentajes de errores activados, las coberturas de detección y recuperación de errores tanto de los mecanismos de tolerancia a fallos como del sistema global, las latencias de propagación, y las latencias de detección y recuperación de los errores propagados. Asimismo, se han llevado a cabo numerosos experimentos variando una serie de parámetros (como la frecuencia de funcionamiento del sistema, la carga de trabajo o la duración de los fallos) para analizar su influencia en el comportamiento del sistema. Con los resultados obtenidos se ha evaluado la efectividad de los mecanismos de tolerancia a fallos del sistema (tanto los de detección como los de recuperación de errores), y en los casos en los que se han detectado deficiencias se han propuesto posibles soluciones.

La validación del controlador TTP/C-C1 (versión 1) se ha realizado en el contexto del proyecto de investigación europeo FIT, y se ha llevado a cabo de manera diferente. En este caso, no se han inyectado fallos sobre el sistema completo, sino sobre algunos componentes. La selección de los componentes "críticos" se ha realizado mediante una serie de experimentos globales, inyectando fallos sobre todo el sistema e identificando los componentes más sensibles a los fallos. A partir de la selección de los componentes críticos, se llevado a cabo una *Eliminación de fallos*, inyectando fallos sobre cada uno de los componentes y calculando las coberturas y latencias correspondientes. El resultado más interesantes que se puede destacar de estos experimentos ha sido la localización de errores de implementación del modelo.

En lo que se refiere a los modelos de los microcontroladores PIC y 8051, se ha analizado su síndrome de error. Sin embargo, este estudio estaba enfocado al estudio de la representatividad de los modelos de fallos en los niveles lógico y RT. Para ello se han inyectado fallos en la lógica combinacional del modelo (nivel lógico), y se han investigado sus efectos en los registros (nivel RT). Se han calculado parámetros como el número y porcentaje de registros afectados, el número y porcentaje de fallos propagados, la multiplicidad de los fallos (o lo que es lo mismo, a cuántos registros es capaz de afectar un solo fallo), y el número y porcentaje de

averías provocadas por los fallos propagados. Además, se ha llevado a cabo una clasificación de los tipos de fallos propagados a los registros. También se ha investigado la influencia de otros factores (la frecuencia de reloj, la carga y la duración de los fallos) en los resultados obtenidos.Entre los resultados más relevantes, se pueden destacar dos: (1) a medida que la frecuencia de funcionamiento aumenta, crecen los porcentajes de fallos propagados y de averías ocasionadas, y (2) los modelos de fallos propagados al nivel RT son, principalmente, *bit-flip* e *indetermination*.

7.7 Trabajo futuro

El objetivo propuesto a corto y medio plazo es la continuación de la línea de trabajo en curso desde diferentes frentes:

1. Completar todos los experimentos de inyección (sobre los modelos de los microcontroladores PIC y 8051) planificados para el estudio de la representatividad de los fallos a nivel RT.
2. Realizar, sobre dichos modelos, los experimentos conducentes a demostrar que omitiendo la inyección de fallos o la observación de la propagación de los errores sobre los registros ocultos causa errores importantes de representatividad de los resultados, justificando la necesidad de complementar las técnicas de inyección más utilizadas en la actualidad con otras que lo permitan, como es el caso de la inyección de fallos en modelos en VHDL.
3. Comparar los resultados obtenidos para ver si se sigue manteniendo una pauta de comportamiento.
4. A medio plazo, se quiere llevar a cabo todo el conjunto de experimentos sobre otros modelos más complejos. En la actualidad se dispone de los modelos sintetizables de dos microprocesadores RISC: LEON-2 y HORUS.

A largo plazo, se pretende continuar aplicando VFIT para validar otros modelos de sistemas tolerantes a fallos complejos.

8 Conclusiones y trabajo futuro

En este capítulo se realiza una recapitulación del trabajo expuesto en la presente tesis. En primer lugar, se extraerán las conclusiones importantes del trabajo desarrollado. También se expondrán las publicaciones a que, directa o indirectamente, ha dado lugar. Por último se abordarán las futuras líneas de trabajo por las que se puede continuar.

8.1 Conclusiones

El presente trabajo de tesis ha abordado diferentes aspectos de la técnica de inyección de fallos mediante simulación de modelos en VHDL: la representatividad de los modelos de fallos, las características generales de la técnica, y las particulares de las diferentes variantes de implementación, y por último, la realización de una herramienta de inyección y su aplicación sobre diferentes modelos.

8.1.1 Representatividad de los modelos de fallos

Como en cualquier técnica de inyección, en la inyección de fallos sobre modelos en VHDL es muy importante que el conjunto de modelos de fallos que se puedan aplicar sea lo más amplio y, sobre todo representativo, como sea posible.

Acerca de este tema, se ha estudiado la bibliografía más importante, prestando especial atención a las publicaciones más recientes, que pronostican sensibles variaciones en dos vertientes principales relacionadas con los fallos en las nuevas tecnologías submicrónicas: los mecanismos de fallo que predominarán en las tecnologías submicrónicas, y las tasas de los fallos transitorios, intermitentes y permanentes.

La considerable reducción de las geometrías de diseño en los circuitos integrados submicrónicos actuales, junto con el fuerte aumento de la frecuencia de funcionamiento van a afectar en gran medida a los tipos de mecanismos de fallo más habituales. Se prevé que el abanico de causas que van a ocasionar fallos en los modernos circuitos integrados se amplíe considerablemente. En particular, los mecanismos que provocan fallos transitorios e intermitentes, van a tener más relevancia, lo cual es especialmente crítico porque estos fallos son difíciles de localizar y tratar.

Elementos de los circuitos integrados como las capas de óxido, difusiones, metalizaciones, etc., se ven afectadas por mecanismos de desgaste (rotura de la capa de óxido, electromigración, defectos de encapsulamiento, etc.) que ocasionan fallos permanentes e intermitentes que, a nivel electrónico, se manifiestan predominantemente como cortocircuitos y circuitos abiertos. Del estudio de los mecanismos de fallo, se han deducido los siguientes modelos de fallos en los niveles lógico y RT: *stuck-at*, *open-line*, *stuck-open*, *indetermination*, *delay*, *short* y *bridging*.

La reducción de las geometrías y de las tensiones de alimentación influyen en los mecanismos de fallo transitorios de dos maneras:

- Disminuyendo la energía necesaria para hacer conmutar los transistores (denominada carga crítica, Q_{crit}). Si se suma el aumento de las frecuencias de funcionamiento, se ha producido una notable mengua del efecto de los mecanismos naturales de enmascara-

miento de fallos de los circuitos digitales (eléctrico, lógico y por ventana temporal de captura).

- A causa de las reducidas distancias entre las conexiones de los chips han aparecido efectos capacitivos, que también se han visto potenciados por las elevadas frecuencias, que afectan a la respuesta temporal de los circuitos.

La disminución de la carga crítica ha provocado que tanto la radiación interna (debida a las partículas α) como la externa (debida a los rayos cósmicos) sean capaces de afectar a los circuitos integrados submicrónicos en mayor medida que en las tecnologías clásicas. Se da el caso de que mecanismos de fallo que en tecnologías tradicionales tenían muy bajo impacto, como la radiación por partículas α y de rayos cósmicos de baja energía (inferior a 1 MeV), en las tecnologías actuales presentan unas tasas de fallo cada vez mayores. Además, a causa de la atenuación de los mecanismos de enmascaramiento, los fallos ocasionados en la lógica combinacional pueden propagarse fácilmente a los elementos de almacenamiento (registros y memorias).

Otro efecto muy importante relacionado con la reducción de la carga crítica es el aumento de la probabilidad de que se produzcan fallos múltiples, sobre todo a causa de la radiación de rayos cósmicos de alta energía.

Para los fallos transitorios se han propuesto, aparte del clásico *bit-flip*, otros modelos de fallos que pueden representar los fallos causados por radiación: *pulse* e *indetermination*. La diferencia entre los modelos *bit-flip* y *pulse* estriba en los diferentes lugares donde se aplican. 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. El modelo *pulse*, en cambio, 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.

8.1.2 Inyección de fallos sobre modelos en VHDL

Antes de abordar el estudio de esta técnica de inyección de fallos, se ha llevado un exhaustivo análisis de la inyección de fallos en general, como método 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). En este sentido, se han destacado sus ventajas respecto a otras técnicas de validación, como las basadas en la observación directa del sistema o en métodos analíticos:

- A causa de las bajas tasas de fallos, principalmente en sistemas tolerantes a fallos, es necesario un elevado tiempo de observación del sistema real para poder medir los valores de las coberturas y latencias con una representatividad estadísticamente aceptable.
- Los métodos analíticos, por su parte, se basan en la resolución de modelos basados en redes de Petri estocásticas, cadenas de Markov, etc., que requieren el conocimiento previo de los valores de las coberturas y latencias para incorporalos en los modelos.

Ya hablando de las diferentes técnicas de inyección de fallos, éstas se pueden clasificar en tres: inyección física (o HWIFI), implementada por *software* (o SWIFI) y mediante simula-

ción. Se ha realizado un profundo estudio del "estado del arte" de todas las técnicas de inyección, describiendo las herramientas y propuestas más importantes.

Además, se ha llevado a cabo una comparación de las diferentes técnicas bajo diferentes puntos de vista, como la sencillez de implementación, su coste (tanto económico como temporal), su interferencia sobre el sistema al que se aplica, etc. La conclusión más destacable extraída de dicha comparación es el hecho de que todas ellas tienen ventajas e inconvenientes. Por ejemplo:

- Las técnicas HWIFI tienen como principal defecto un elevado coste (bien de implementación o de preparación), además de modelos de fallos limitados. Como ventaja ofrecen una elevada precisión (equivalencia entre los fallos inyectados y los mecanismos de fallo reales). Otro aspecto interesante de estas técnicas, que comparten con las SWIFI, es que se aplican sobre el sistema real (o un prototipo del mismo), con su carga de trabajo real.
- Las técnicas SWIFI son muy populares debido a su reducido coste de implementación y a su facilidad de automatización. Por contra, adolece de bajos niveles de genericidad (o independencia de los sistemas), accesibilidad (facilidad de acceso a los puntos de inyección) y controlabilidad (capacidad de control de los atributos de los fallos inyectados).
- La principal ventaja de lass 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. Además, ofrecen altos niveles de genericidad, accesibilidad, controlabilidad y capacidad de automatización. Su principal inconveniente es el elevado coste temporal de la realización de la inyección.

Por tanto, ninguna de las técnicas puede ser considerada como definitiva. Antes al contrario, las tres técnicas deberían emplearse de forma complementaria en diferentes fases del diseño y desarrollo de los sistemas.

Como quiera que el Grupo de Sistemas Tolerantes a Fallos, al que pertenece el autor, ya había desarrollado sendas herramientas de inyección de fallos HWIFI y SWIFI, se propuso la implementación de una herramienta de inyección mediante simulación. En particular, se optó por la simulación de modelos en VHDL, debido por un lado a la gran aceptación que este lenguaje de descripción de *hardware* tiene en la comunidad científica (por ser un estándar del IEEE, porque permite realizar descripciones multinivel, etc.), y por otro, a que la sintaxis de VHDL ofrece algunos mecanismos útiles para la inyección de fallos. Además de las razones expuestas anteriormente, existen simuladores comerciales que permiten su integración en una herramienta de inyección.

La presente tesis parte de algunos trabajos previos sobre simulación en VHDL realizados en el seno del grupo (véase la tesis de D. Daniel Gil Tomás), en la que incluso se realizó un prototipo de herramienta de inyección de fallos mediante simulación de modelos en VHDL, implementado también en VHDL.

Puesto el objetivo de esta tesis era completar algunos aspectos de la mencionada técnica de inyección de fallos (así como desarrollar una nueva herramienta de inyección), se le ha dedicado una atención especial. En efecto, además de exponer las diferentes subtécnicas existentes, se han elegido un subconjunto de ellas para su implementación en la herramienta de inyección de fallos. Las técnicas elegidas han sido la basada en las órdenes del simulador (per-

teneciente 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. Este hecho se hace más destacable en función de la implementación particular de cada técnica. Así:

- La técnica de órdenes del simulador se basa simplemente en la utilización de órdenes del simulador para controlar la simulación del modelo y alterar el valor y/o la temporización de los elementos internos del modelo. Es, con diferencia, la más sencilla de implementar y automatizar. Tiene como inconveniente que el conjunto de modelos de fallos que se pueden aplicar es dependiente de las capacidades del simulador, y suele ser reducido.
- La técnica basada en perturbadores se aplica a modelos con descripciones estructurales. Consiste en interponer, en las conexiones de los componentes de un nivel de jerarquía, unos componentes especiales, denominados perturbadores, cuya misión es modificar los valores y/o la temporización de las señales interceptadas. Los perturbadores añadidos se controlan mediante entradas especiales que determinan si se inyecta o no un fallo, y en caso positivo, el modelo de fallo inyectado. Esta técnica presenta cierta complejidad de implementación y automatización, especialmente en lo que se refiere a la elaboración de los componentes perturbadores. A cambio, permite aplicar un conjunto de modelos de fallos mayor que en la técnica basada en las órdenes del simulador.
- La técnica basada en mutantes es adecuada principalmente para modelos con descripciones comportamentales (o estructurales con descripciones comportamentales en los niveles más bajos de la jerarquía). Se basa en realizar múltiples versiones modificadas (mutadas) del código que representan un fallo (es decir, una única modificación del código). Las modificaciones pueden afectar a diversas sentencias del código VHDL. La inyección de un fallo consiste en alternar la simulación de la versión original del sistema con una alterada. 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 espacio en disco, necesario para almacenar las múltiples versiones mutadas del modelo. La ventaja principal de esta técnica es una gran variedad de modelos de fallos, muy superior a los de las otras dos.

Tras estudiar la problemática existente en las tres técnicas, en particular en perturbadores y mutantes, se han propuesto nuevas alternativas de implementación de estas dos últimas, para solventar las dificultades que se presentan.

En el caso de los perturbadores, se han implementado cuatro modelos de perturbadores para conectar líneas de tipo digital, que simplifican notablemente otras propuestas de conjuntos de modelos. No sólo se ha propuesto una única implementación de los cuatro modelos, sino diversas versiones (de diferente complejidad) que permiten inyectar distintos tipos de fallos atendiendo a su multiplicidad. En este sentido, se tiene desde una versión simple, con la que sólo se pueden inyectar fallos simples, hasta una altamente sofisticada que permite inyectar fallos múltiples en el espacio y en el tiempo. También se ha propuesto una extensión de los cuatro modelos generales a líneas de cualquier tipo, y sobre todo, un método de automatización de la inserción de los perturbadores en el modelo y de la realización de la inyección.

En cuanto a los mutantes, se ha desarrollado una implementación que pretende resolver algunos de los problemas que presenta dicha técnica. El método consiste en realizar una única

versión del modelo que contenga todas las mutaciones seleccionadas previamente. La activación y desactivación de una determinada mutación (fallo) se realiza mediante entradas externas, de forma similar a como se hace en los perturbadores, eliminando de este modo el problema de la sincronización. La reducción del espacio de disco ocupado también es notable, si bien los ficheros conteniendo las versiones mutadas del código pueden tener un tamaño desmesurado. Al igual que para los perturbadores, también se ha propuesto un método para la automatización de la realización del modelo mutado.

En lo referente al "estado del arte" de la inyección de fallos mediante simulación de modelos en VHDL, se han comentado los trabajos más relevantes el tema (en cuanto a técnicas y herramientas), así como otros que, aunque pertenecientes a otros campos de investigación, tratan algunos aspectos relacionados con la inyección de fallos mediante simulación de modelos en VHDL, como modelos de fallos, propuestas de implementación de perturbadores y mutantes, etc.

Al investigar las herramientas existentes, se ha detectado la ausencia de herramientas que funcionen sobre PC (de hecho, todas las existentes lo hacen sobre estaciones de trabajo), así como de implementaciones de las técnicas basadas en perturbadores y mutantes.

8.1.3 Realización de la herramienta de inyección

El trabajo desarrollado ha culminado con la realización de VFIT, una herramienta que funciona sobre PC bajo Windows®. La realización de un experimento de inyección se divide en tres fases:

1. Configuración, donde se establecen los parámetros de inyección y los tipos de resultados que se esperan.
2. Simulación, en la que se simula el modelo del sistema bajo estudio sin y con fallos, obteniéndose una serie de trazas de las diferentes simulaciones.
3. Análisis y extracción de resultados. En esta fase se comparan las trazas de las simulaciones con fallos con la de la simulación sin fallos.

Las principales características de la herramienta desarrollada son:

- Gracias a una potente interfaz con el usuario, es independiente del modelo, y completamente automatizada. Una vez finaliza la configuración de los parámetros, las fases de simulación y análisis se ejecutan de forma autónoma.
- Permite inyectar fallos utilizando las tres técnicas de inyección estudiadas. No obstante, aunque en el prototipo actual de VFIT sólo se incorpora la técnica basada en órdenes del simulador, la automatización e inclusión de las otras dos técnicas está en fase de desarrollo.
- Permite inyectar un amplio conjunto de modelos de fallos en las tres técnicas de inyección.
- En función de los tipos de resultados esperados indicados en la fase de configuración, el análisis puede ser de dos clases:
  - ⇨ Estudio del síndome de error. Este tipo de análisis consiste en (a partir de sendas clasificaciones de los fallos inyectados y de los errores propagados) analizar el efecto de los fallos inyectados en el sistema (tanto en cuanto a errores propagados

como en averías ocasionadas), así como medir las latencias de propagación de los diferentes errores propagados. Este estudio se suele aplicar principalmente a sistemas no tolerantes a fallos, y su objetivo es seleccionar los mecanismos de tolerancia a fallos más adecuados para su inclusión en el sistema.

  - ⇨ Validación de un sistema tolerante a fallos. En esta caso, el estudio se orienta al cálculo de una serie de parámetros que permiten cuantificar la bondad de los mecanismos de tolerancia a fallos del sistema. Principalmente son las coberturas de detección y recuperación de errores, y los tiempos de latencia en la detección y recuperación de los errores.

- Permite realizar las fases de simulación y análisis de forma paralela (que no distribuida; ése es un aspecto en el que se va a trabajar en el futuro), gracias al concepto de sesión.

8.1.4 Aplicación de la herramienta

VFIT se ha empleado (y se sigue utilizando) para inyectar fallos sobre diversos modelos de sistemas en VHDL. Las campañas de inyección realizadas pertenecen tanto al ámbito de proyectos de investigación europeos (DBench, FIT) como a otras líneas de trabajo del GSTF. Asimismo, los diferentes experimentos de inyección llevados a cabo tienen objetivos variados, en función del modelo y del entorno de trabajo al que pertenece.

Entre los modelos sobre los que se ha aplicado la herramienta, cabe destacar:

- Un sistema computador tolerante a fallos basado en el microprocesador MARK2. El objetivo de los experimentos realizados era la validación de este modelo académico, calculando principalmente las coberturas de detección y recuperación de errores, así como las latencias de propagación, detección y recuperación de errores.
- Un controlador de comunicaciones tolerante a fallos, basado en la arquitectura TTA (Time-Triggered Architecture): TTP/C. Los experimentos sobre este modelo se han llevado a cabo en el contexto del proyecto europeo FIT (Fault Injection for TTA, IST-1999-10748), cuyo objetivo era la validación del controlador, en este caso buscando deficiencias en los mecanismos de tolerancia a fallos.
- Dos microcontroladores comerciales: PIC y 8051. Se ha estudiado el síndrome de error de estos modelos, con el objetivo final de estudiar la representatividad de los modelos de fallos a nivel RT. Los experimentos realizados sobre estos modelos pertenecen a una de las líneas de investigación del GSTF más recientes, y aún está en curso. Se pretende completar inyectando fallos sobre otros modelos de microprocesadores más complejos (LEON-2, HORUS).

8.2 Resultados de investigación

Como consecuencia del trabajo desarrollado para la realización de la presente tesis se han publicado un número considerable de artículos en revistas y de ponencias en congresos, todos ellos de relevancia en el campo de la inyección de fallos en VHDL. Estas publicaciones, a su vez, han contribuido en la consecución de un tramo de investigación por parte del autor.

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 enumeran, en primer lugar, los trabajos más relevantes en relación con esta tesis que se han publicado. En segundo lugar, se exponen algunas de las publicaciones donde se citan trabajos en los que ha participado el autor.

8.2.1 Publicaciones

- J. Gracia, D. Gil, J.C. Baraza, P.J. Gil, "Application of Different VHDL-Based Fault Injection Techniques to the Validation of a Fault Tolerant Microcomputer System", en *Workshops and Abstracts of the International Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8)*, Nueva York (Nueva York, EE.UU.), pp. B-54–B-55, Junio 2000.
- [Gil *et al.* 2000] D. Gil, J. Gracia, J.C. Baraza, P.J. Gil, "A Study of the Effects of Transient Fault Injection into the VHDL Model of a Fault Tolerant Microcomputer System", en *Proceedings 6th IEEE International On-Line Testing Workshop (IOLTW 2000)*, Palma de Mallorca, pp. 73-79, Julio 2000.
- J.C. Baraza, J. Gracia, D. Gil, P.J. Gil, "A Prototype of a VHDL-Based Fault Injection Tool", en *Proceedings 2000 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2000)*, Yamanashi (Japón), pp. 396-404, Octubre 2000.
- J. Gracia, J.C. Baraza, D. Gil, P.J. Gil, "A Study of the Experimental Validation of Fault-Tolerant Systems Using Different VHDL-Based Fault Injection Techniques", en *Proceedings 7th IEEE International On-Line Testing Workshop (IOLTW 2001)*, Taormina (Italia), pág. 140, Junio 2001.
- [Gracia *et al.* 2001b] J. Gracia, J.C. Baraza, D. Gil, P.J. Gil, "Comparison and Application of Different VHDL-Based Fault Injection Techniques", en *Proceedings 2001 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2001)*, San Francisco (California, EE.UU.), pp. 233-241, Octubre 2001.
- J.C. Baraza, J. Gracia, D. Gil, P.J. Gil, "A Prototype of a VHDL-Based Fault Injection Tool: Description and Application", *Journal of Systems Architecture*, ISSN 1383-7621, 47(10):847-867, Abril 2002.
- [DBench 2002] P.J. Gil *et al.*, "Fault Representativeness", *Deliverable ETIE2 of Dependability Benchmarking Project (DBench), IST-2000-25245*, Julio 2002.
- J. Gracia, D. Gil, J.C. Baraza, P.J. Gil, "Using VHDL-Based Fault Injection to exercise Error Detection Mechanisms in the Time-Triggered Architecture", en *Proceedings 2002 Pacific Rim International Symposium on Dependable Computing (PRDC 2002)*, Tsukuba (Japón), pp. 316-320, Diciembre 2002.
- D. Gil, J. Gracia, J.C. Baraza, P.J. Gil, "Study, Comparison and Application of different VHDL-Based Fault Injection Techniques for the Experimental Validation of a Fault-Tolerant System", *Microelectronics Journal*, 34(1):41-51, Enero 2003.
- J. Gracia, J.C. Baraza, D. Gil, P.J. Gil, "Early Diagnosis of Hard Real-Time Fault-Tolerant Embedded Systems", en *Proceedings 6th International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2003)*, Poznań (Polonia), pp.157-164, Abril 2003.

- J.C. Baraza, D. de Andrés, D. Gil, P.J. Gil, "Técnicas de inyección de fallos basadas en VHDL". Tutorial presentado en las III Jornadas sobre Computación Reconfigurable y Aplicaciones (JCRA 2003), Cantoblanco (Madrid), Setiembre 2003. Publicado en E. Boemo et al., *Computación Reconfigurable & FPGAs*, ISBN 84-600-9928-8, pp.621-635.
- D. Gil, J. Gracia, J.C. Baraza, P.J. Gil, "Using VHDL-based fault injection for the early diagnosis of a TTP/C controller", *IEICE Transactions on Dependable Computing*, Aceptado, 2003.
- D. Gil, J.C. Baraza, J. Gracia, P.J. Gil, "VHDL simulation-based fault injection techniques", capítulo 7 en *Fault injection techniques and tools for VLSI reliability evaluation*, Aceptado, 2003.

8.2.2 Referencias

- R. Leveugle, K. Hadjiat, "Multi-Level Fault Injection Experiments Based on VHDL Descriptions: A Case Study", en *Proceedings 8th IEEE International On-Line Testing Workshop (IOLTW 2002)*, Isla de Bendor (Francia), pp. 107-111, Julio 2002.

  Artículo citado: [Gil *et al.* 2000].

- M. Rebaudengo, M. Sonza Reorda, M. Violante, "Analysis of SEU Effects in a Pipelined Processor", en *Proceedings 8th IEEE International On-Line Testing Workshop (IOLTW 2002)*, Isla de Bendor (Francia), pp. 112-116, Julio 2002.

  Artículo citado: [Gracia *et al.* 2001b].

- G.C. Cardarilli, F. Kaddour, A. Leandri, M. Ottavi, S. Pontarelli, R. Velazco, "Bit Flip Injection in Processor-Based Architectures: A Case Study", en *Proceedings 8th IEEE International On-Line Testing Workshop (IOLTW 2002)*, Isla de Bendor (Francia), pp. 117-127, Julio 2002.

  Artículo citado: [Gracia *et al.* 2001b].

- 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 *Proceedings 2003 International Conference on Dependable Systems and Networks (DSN 2003)*, pp. 83-88, San Francisco (California, EE.UU.), Junio 2003.

  Artículo citado: [Gracia *et al.* 2001b].

- Ö. Askerdal, "On impact and tolerance of data errors with varied duration in microprocessors", Tesis doctoral, Technical University of Chalmers, 2003.

  Artículos citados: [DBench 2002, Gracia *et al.* 2001b].

8.3 Trabajo futuro

Parte del trabajo desarrollado para la realización de la presente tesis se puede continuar desde diferentes líneas de trabajo abordadas por el Grupo de Sistemas Tolerantes a Fallos. Este es el caso de la representatividad de los modelos de fallos, los modelos y métodos de implementación de las técnicas de inyección (perturbadores y mutantes), y la propia herramienta.

En cuanto a la representatividad de los modelos de fallos, está previsto proseguir el trabajo desde tres frentes:

- Utilizar herramientas de simulación electrónica para estudiar el efecto de los fallos transitorios en nuevas tecnologías, o el análisis de las características temporales de los fallos transitorios, en particular su duración.
- Inyectar fallos en diversos modelos en VHDL de microcontroladores y microcoprocesadores para comprobar empíricamente las limitaciones de las técnicas SWIFI, vistas desde dos puntos de vista:
  - ⇨ Inyectando fallos en la lógica combinacional y estudiando las perturbaciones ocasionadas en los registros ocultos (no accesibles vía *software*).
  - ⇨ Inyectando fallos en los registros, distinguiendo entre las averías producidas por fallos en registros accesibles por *software* y en registros ocultos.
- Continuar el estudio de la representatividad de los modelos de fallos propuestos en el nivel RT, así como pensar en conectar con otros niveles superiores (algorítmico, etc.). De este modo se podría deducir un conjunto de modelos de fallos representativos para mutantes comportamentales.

En el caso de la implementación de las técnicas de inyección, se prevé probar exhaustivamente los modelos de perturbadores y mutantes propuestos sobre los modelos de sistemas que el grupo ya dispone, para probar su efectividad.

Por último, está previsto ampliar las capacidades de VFIT desde diferentes puntos de vista:

- Integrar las técnicas de inyección de fallos basadas en perturbadores y mutantes, automatizando su generación, inserción y aplicación.
- Permitir la inyección de fallos múltiples, tanto desde el punto de vista temporal como del espacial.
- Permitir realizar las fases de simulación y análisis de manera distribuida (principalmente la primera), siendo la propia herramienta quien se encargue del reparto de tareas a los diferentes PC de una red.
- Ampliar el conjunto de funciones de distribución de probabilidad para generar el instante de inyección y la duración del fallo.
- Mejorar la interfaz herramienta-usuario para incorporarle nuevas facilidades de manejo (de cara al usuario) y aumentar sus posibilidades de trabajo (es decir, de los resultados que se pueden extraer en la fase de análisis): generación automática de tablas, gráficas, histogramas, etc.

- Investigar e implementar otras técnicas para optimizar el coste temporal del proceso global: inyección-simulación-análisis.
- Comparar resultados de la inyección de fallos con otras técnicas (HWIFI, SWIFI, etc.) sobre el mismo sistema.
- Profundizar en el estudio de la influencia de la carga de trabajo (*workload*).

Apéndice A Funciones de distribución de fallos

En este apéndice se describen las funciones de distribución más utilizadas para expresar el instante en que se produce un fallo físico en un sistema: Exponencial, Uniforme, Normal, Lognormal, Rayleigh, Weibull, y Gamma. Algunas de ellas tienen abundantes evidencias empíricas, a partir de análisis de sistemas reales. Otras, en cambio, presentan características teóricas que las hacen atractivas. Por otro lado, estas funciones también se pueden aplicar para representar la duración de los fallos transitorios e intermitentes[89].

Como introducción a la descripción de las funciones de distribución, en el apartado A.1 se exponen algunos conceptos básicos. En los apartados A.2 a A.8 se describen las funciones de distribución más comunes.

Las fuentes bibliográficas en que se basa este apéndice son [Meyer 1986, Amerasekera y Najm 1997, Trivedi 1982]

A.1 Conceptos básicos

El instante de fallo de un sistema, denotado por T, se puede considerar como una variable aleatoria continua con una función de densidad de probabilidad f(t).

La función de densidad de probabilidad está relacionada, a su vez, con otros conceptos estadísticos, como:

- La función de fiabilidad, R(t), que es la probabilidad de que el sistema funcione correctamente en el intervalo [0, t]:

$$R(t) = P\{T > t\}$$

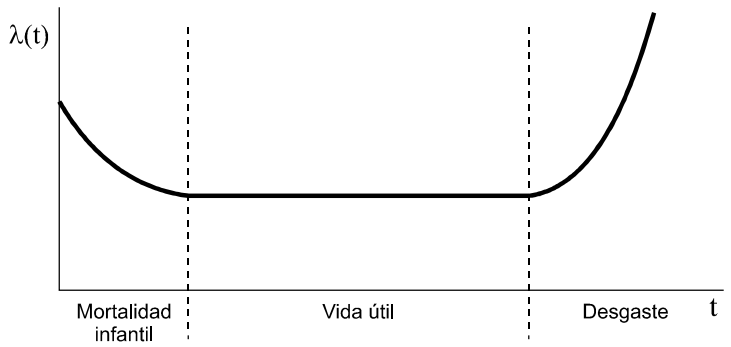

Figura A.1: Curva de la bañera.

- La tasa de fallos[90], λ(t). La representación de la tasa de fallos (físicos) en función del tiempo suele tener la forma mostrada en la Figura A.1. Dicha curva recibe el nombre de

[89] En realidad, la problemática de la duración de los fallos transitorios e intermitentes es un tema muy poco estudiado [DBench 2002], y se trata de un tema abierto.

[90] También conocida como tasa de averías (del inglés *failure rate*). Tradicionalmente, estas funciones se han utilizado para especificar el instante en que se producen averías, ocasionadas por fallos permanentes en el sistema. Sin embargo, se ha demostrado empíricamente que la ocurrencia de fallos no permanentes (es decir, transitorios e intermitentes) también responde con estas distribuciones.

"curva de la bañera[91]. Como se puede ver en la figura, la curva se puede dividir en tres zonas de comportamiento diferenciado:

- ⇨ *Mortalidad infantil*, que representa a los fallos tempranos que se producen por defectos de fabricación.
- ⇨ *Vida útil*, que se corresponde con el funcionamiento normal del sistema.
- ⇨ *Desgaste*, que considera los fallos ocasionados por el desgaste de los materiales.

Puede demostrarse que la relación existente entre la función de densidad de probabilidad, la función de probabilidad y la tasa de fallos es:

$$\lambda(t) = \frac{f(t)}{R(t)}$$

A.2 Distribución Exponencial

Esta es una de las distribuciones de fallos más comunes. Se caracteriza por tener una tasa de fallos (λ) constante. Las funciones f(t), R(t) y λ(t) son:

$$f(t) = \alpha e^{-\alpha t}, \; t > 0$$

$$R(t) = e^{-\alpha t}$$

$$\lambda(t) = \alpha, \; \alpha \text{ constante}$$

En la Figura A.2 se muestran las funciones f(t) y λ(t).

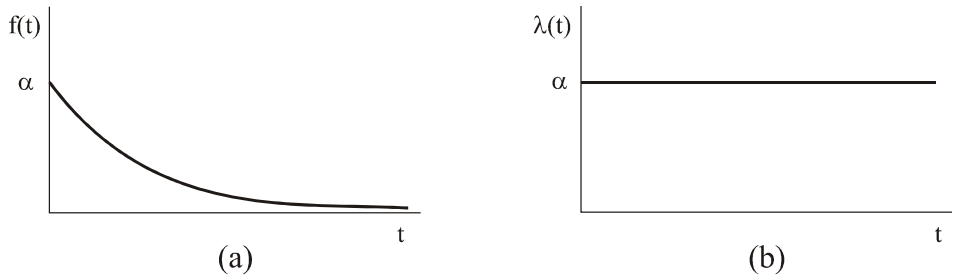

Figura A.2: Distribución Exponencial. (a) Función de densidad de probabilidad. (b) Tasa de fallos.

La asunción de que la tasa de fallos es constante hace que esta función sirva para modelar la zona de vida útil de la curva de la bañera, donde se supone que no afecta el desgaste por el uso del sistema.

[91] En inglés, *bathtub curve*.

A.3 Distribución Uniforme

Las funciones f(t), R(t) y λ(t) son:

$$f(t) = \begin{cases} \dfrac{1}{t_1 - t_0}, t_0 \le t \le t_1 \\ \\ 0, \text{ en otro caso} \end{cases}$$

$$R(t) = \frac{t_1 - t}{t_1 - t_0}, t_0 \le t \le t_1$$

$$\lambda(t) = \frac{1}{t_1 - t_0}, t_0 \le t \le t_1$$

En la Figura A.3 se muestran las funciones f(t) y λ(t). En ellas se observa que función de densidad de probabilidad, f(t), es constante en el intervalo $[t_0, t_1]$, y cero fuera de él. En lo que se refiere a la tasa de fallos, es creciente con t en el intervalo $[t_0, t_1]$.

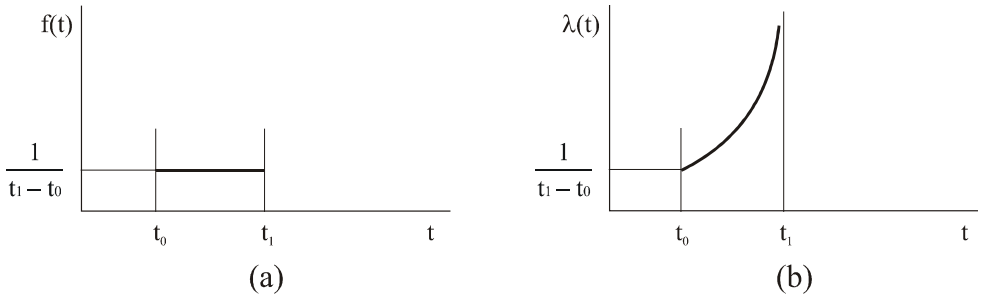

Figura A.3: Distribución Uniforme. (a) Función de densidad de probabilidad. (b) Tasa de fallos.

A.4 Distribución Normal

La distribución Normal (también conocida como Gaussiana) es una de las distribuciones más conocidas y utilizadas. Se suele emplear para modelar muchos fenómenos naturales. En el estudio de la fiabilidad, la distribución Normal puede ser útil para modelar la zona de desgaste.

La función de densidad de probabilidad es:

$$f(t) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{t-\mu}{\sigma}\right)^2}, \quad -\infty < t < +\infty$$

En forma abreviada, $f(t) \equiv N(\mu, \sigma^2)$, donde μ es la *media* y σ^2 es la *varianza*.

Las expresiones para λ(t) y R(t) no se pueden evaluar de forma analítica, y se utilizan métodos de integración numérica que están tabulados.

En la Figura A.4 se muestran las funciones f(t) y λ(t).

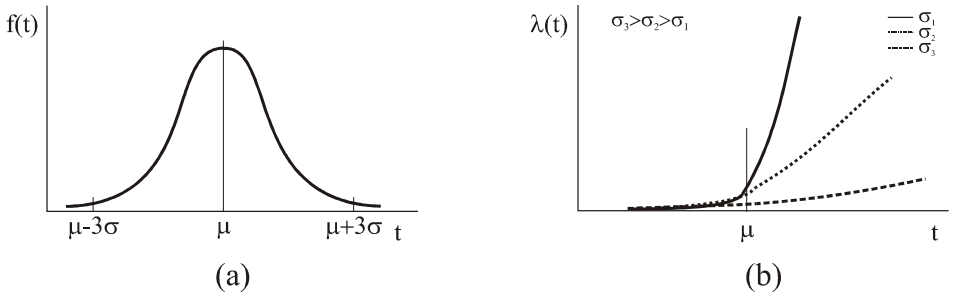

Figura A.4: Distribución Normal. (a) Función de densidad de probabilidad. (b) Tasa de fallos.

La forma de campana de f(t) indica que la mayor parte de los fallos ocurren alrededor de la media (μ), y el número de fallos disminuye a medida que |T-μ| aumenta. Así, en el rango de valores comprendidos entre μ-3σ y μ+3σ tienen lugar el 99.72 % de los fallos.

De la curva de la tasa de fallos se observa que es creciente con t, por lo que se puede utilizar para modelar la zona de desgaste. Por el contrario, λ(t) es decreciente con σ.

Este tipo de distribución puede ser particularmente interesante cuando se considera que el instante de inyección es la suma de varios factores aleatorios, ninguno de los cuales es predominante sobre los demás. Según el Teorema del Límite Central, la distribución de la suma de un elevado número de variables aleatorias independientes tiende a ser Normal, sin importar los tipos de las distribuciones individuales.

A.5 Distribución Lognormal

Si ln T sigue una distribución Normal, se dice que T tiene una distribución Lognormal, cuya función de distribución de probabilidad es:

$$f(t) = \frac{1}{t\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{\ln t - \mu}{\sigma}\right)^2}, \quad 0 < t < +\infty,$$

donde μ es la *media* de ln T y σ^2 es su *varianza*.

Como en la distribución Normal, la función de fiabilidad y la tasa de fallos se calculan mediante tablas.

En la Figura A.5 se muestran algunas curvas típicas para f(t) y λ(t). En general, la tasa de fallos puede ser creciente o decreciente, según el intervalo de tiempo considerado.

Esta distribución puede ser adecuada cuando el instante de fallo se puede especificar como el producto de varios factores aleatorios. Si se tiene que

$$T = T_1 \cdot T_2 \cdot \ldots \cdot T_n$$

entonces

$$\ln T = \ln T_1 + \ln T_2 + \ldots + \ln T_n$$

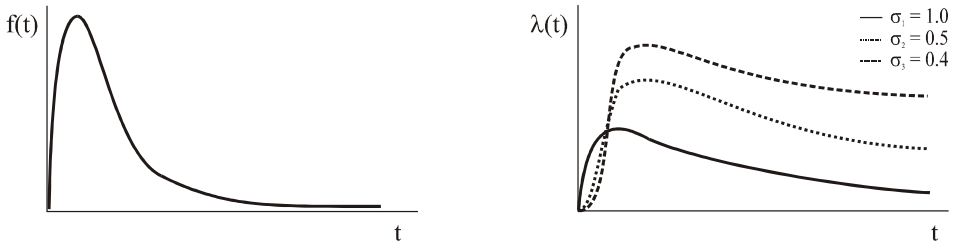

Figura A.5: Distribución Lognormal. (a) Función de densidad de probabilidad. (b) Tasa de fallos.

Por tanto, si las variables aleatorias T_i son (aproximadamente) independientes, entonces ln T es (aproximadamente) Normal para un valor elevado de *n* (o lo que es lo mismo, T es Log-normal).

Como ejemplo, se puede considerar el caso de que haya varios mecanismos de fallo que actúen de manera interrelacionada para provocar el fallo, de manera que el efecto conjunto sea el producto de los efectos individuales (como difusión, temperatura, sobretensión, etc.).

La distribución Lognormal se utiliza cada vez más en aplicaciones de fiabilidad de circuitos integrados.

A.6 Distribución Rayleigh

Se caracteriza porque la tasa de fallos crece de forma lineal. Las funciones f(t), R(t) y λ(t) son:

$$f(t) = kte^{\frac{-kt^2}{2}}$$

$$R(t) = e^{\frac{-kt^2}{2}}$$

$$\lambda(t) = kt, k > 0$$

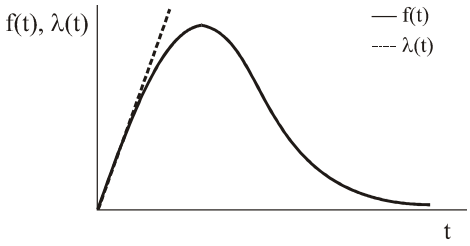

Figura A.6: Distribución Rayleigh.

En la Figura A.6 se muestra la forma típica de las funciones f(t) y λ(t). Como se puede comprobar, tiene una tasa de fallos creciente, por lo que se puede utilizar para modelar la fase de desgaste. Además, es una función muy sencilla de utilizar por disponer de un solo parámetro.

A.7 Distribución Weibull

Esta distribución es muy utilizada en relación con la fiabilidad. Tiene dos parámetros, denominados *factor de escala* (α) y *parámetro de forma* (β), ambos positivos. Las funciones f(t), R(t) y λ(t) son:

$$f(t) = \alpha^{\beta}\beta t^{\beta-1} e^{-(\alpha t)^{\beta}}$$

$$R(t) = e^{-(\alpha t)^{\beta}}$$

$$\lambda(t) = \alpha^{\beta}\beta t^{\beta-1}$$

Esta función es muy versátil, y se puede emplear para proporcionar una gran variedad de formas. Se pueden destacar dos casos particulares:

- Si β = 1, se obtiene una distribución Exponencial con λ = α.
- Si β = 2, se obtiene una distribución Rayleigh con $k = 2\alpha^2$.

El parámetro de forma (β) permite utilizar la distribución Weibull para modelar diferentes situaciones en el ciclo de vida de los sistemas:

- Si 0 < β < 1, la tasa de fallos es decreciente, lo cual permite modelar la zona de mortalidad infantil, o fallos en el *software* que se subsanan.
- Si β = 1, la tasa de fallos es constante, y permite modelar la zona de vida útil (como la Exponencial con λ = α a la que equivale).
- Si 1 < β < +∞, la tasa de fallos es creciente, con lo que se puede modelar la zona de desgaste.

La Figura A.7 muestra las curvas de las funciones f(t) y λ(t) para diferentes valores de β, con α = 1.0.

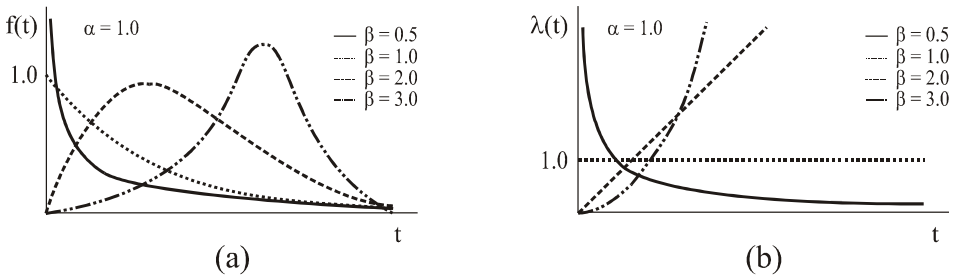

Figura A.7: Distribución Weibull. (a) Función de densidad de probabilidad. (b) Tasa de fallos.

La distribución Weibull representa un modelo apropiado en caso de que, de entre los varios mecanismos de fallo que pueden influir en el instante del fallo, exista alguno predominante o "más grave", como lo sería la rotura de la capa *thinox* en los transistores MOS.

A.8 Distribución Gamma

Esta distribución, al igual que la Weibull, tiene dos parámetros, también denominados *factor de escala* (α) y *parámetro de forma* (β), ambos positivos. Las funciones f(t) y $\lambda(t)$ son:

$$f(t) = \frac{\alpha^{\beta} t^{\beta-1}}{\Gamma(\beta)} e^{-\alpha t}$$

$$\lambda(t) = \frac{1}{\int_0^{\infty} \left(1 + \frac{x}{t}\right)^{\beta-1} e^{-\alpha x} dx}$$

La función $\Gamma(\beta)$ se denomina función gamma, y se expresa como:

$$\Gamma(x) = \int_0^{\infty} t^{x-1} e^{-t} dt, \; x > 0$$

La Figura A.8 muestra varias curvas de la tasa de fallos, para diferentes valores de β. En ella se puede comprobar que el parámetro de forma (β) sirve para modelar la curva, pudiéndose observar los siguientes casos:

- Si $0 < \beta < 1$, la tasa de fallos es decreciente, y permite modelar la zona de mortalidad infantil.
- Si $\beta = 1$, la tasa de fallos es constante e igual a α, y permite modelar la zona de vida útil.
- Si $1 < \beta < +\infty$, la tasa de fallos es creciente.

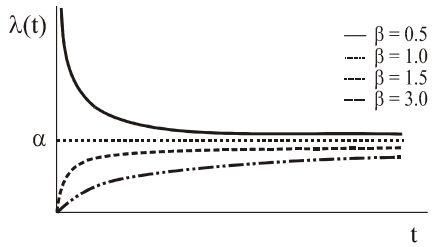

Figura A.8: Tasa de fallos de la distribución Gamma.

Bibliografía

[Aftabjahani y Navabi 1997] S.A. Aftabjahani y Z. Navabi, "Functional Fault Simulation of VHDL Gate Level Models", en *Proceedings 1997 VHDL International Users' Forum (VIUF'97)* pp. 18-23, Arlington (Virginia, EE.UU.), Octubre 1997.

[Aidemark *et al.* 2001] J. Aidemark, J. Vinter, P. Folkesson, J. Karlsson, "GOOFI: Generic Object-Oriented Fault Injection Tool", en *Proceedings 2001 International Conference on Dependable Systems and Networks (DSN 2001)*, pp. 83-88, Göteborg (Suecia), Julio 2001.

[Amendola *et al.* 1996] A. M. Amendola, A. Benso, F. Corno, L. Impagliazzo, P. Marmo, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, "Fault Behavior Observation of a Microprocessor System through a VHDL Simulation-Based Fault Injection Experiment", en *Proceedings 1996 European Design Automation Conference with EURO-VHDL (EURO-DAC'96/EURO-VHDL'96)*, pp. 536-541, Ginebra (Suiza), Setiembre 1996.

[Amendola *et al.* 1997] A.M. Amendola, L. Impagliazzo, P. Marmo, F. Poli, "Experimental Evaluation of Computer-Based Railway Control Systems", en *Proceedings 27th International Symposium on Fault-Tolerant Computing (FTCS-27)*, pp. 380-384, Seattle (Washington, EE.UU.), Junio 1997.

[Amerasekera y Najm 1997] E.A. Amerasekera y F.N. Najm, "Failure Mechanisms in Semiconductor Devices", 2nd edition, ISBN 0-471-95482-9, John Wiley & Sons, 1997.

[Anelli 2000] G.M. Anelli, "Conception et Caracterisation de Circuits Integrés Résistants aux Radiations pour les Detecteurs de Particules du LHC en Technologies CMOS Submicroniques Profondes", *Tesis Doctoral (en inglés)*, Groupe de Microélectronique du Laboratoire Européen pour la Recherche Nucléaire (CERN), Institut National Polytechnique de Grenoble, Grenoble (Francia), Diciembre 2000.

[Arlat 1990] J. Arlat, "Validation de la Sûreté de Fonctionnement par Injection de Fautes. Méthode - Mise en Oeuvre – Application", *Thése présentée à L'Institut National Polytechnique de Toulouse*, LAAS Report nº 90-399, Diciembre 1990.

[Arlat 1992] Arlat J., "Fault injection for the experimental validation of fault-tolerant systems", en *Proceedings 1992 IEICE Workshop on Fault-Tolerant Systems*, pp. 33-40, Kyoto (Japón), Junio 1992.

[Arlat *et al.* 1984] J. Arlat, P. Blanquart, J.C. Laprie, "Sur la Certification des systèmes informatiques: le projet EVE - Application au poste d'aiguillage informatisé", en *Actes 4ème Colloque International Fiabilité et Maintenabilité*, Perros-Guirec et Trégastel, pp. 650-656, Mayo 1984.

[Arlat *et al.* 1990] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.C Fabre, J.C. Laprie, E. Martins, D. Powell, "Fault injection for dependability validation: a methodology and some applications", *IEEE Transactions on Software Engineering*, 16(2):166-182, Febrero 1990.

[Armstrong 1989] J.R. Armstrong, "Chip-Level Modeling with VHDL", 2ª ed., ISBN 0-131-33190-6, Ed. Prentice Hall, 1989.

[Armstrong *et al.* 1992] J.R. Armstrong, F.-S. Lam, P.C. Ward, "Test generation and Fault Simulation for Behavioral Models", en *Performance and Fault Modelling with VHDL*, pp. 240-303, J.M. Schoen ed., Englewood Cliffs, Prentice-Hall, 1992.

[Ashenden 2002] P.J. Ashenden, "The Designer's Guide to VHDL", 2nd edition, ISBN 1-55860-674-2, Morgan-Kaufmann Publishers, 2002.

[Avresky *et al.* 1992] D. Avresky, J. Arlat, J.C. Laprie, Y. Crouzet, "Fault injection for the formal testing of fault tolerance", en *Proceedings 22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, pp. 345-354, Boston (Massachusetts, EE.UU.), Julio 1992.

[Aylor *et al.* 1990] J.H. Aylor, R.D. Williams, R. Waxman, B.W. Johnson, R.L. Blackburn, "A Fundamental Approach to Uninterpreted/Interpreted Modeling of Digital Systems in a Common Simulation Environment", *Technical Report 900724.0*, University of Virginia, 1990.

[Aylor *et al.* 1992] J.H. Aylor, R. Waxman, B.W. Johnson, R.D. Williams, "The Integration of Performance and Functional Modeling in VHDL", en [Shoen 1992], Capítulo 2, pp. 22-145.

[Baraza 1999] J.C. Baraza. "Diseño de una herramienta de inyección de fallos sobre modelos en VHDL", *Trabajo de doctorado de 6 créditos*, Setiembre 1999.

[Baraza *et al.* 2000] J.C. Baraza, J. Gracia, D. Gil, P.J. Gil, "A Prototype of a VHDL-Based Fault Injection Tool", en *Proceedings 2000 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2000)*, Yamanashi (Japón), pp. 396-404, Octubre 2000.

[Baraza *et al.* 2002] J.C. Baraza, J. Gracia, D. Gil, P.J. Gil, "A Prototype of a VHDL-Based Fault Injection Tool: Description and Application", *Journal of Systems Architecture*, ISSN 1383-7621, 47(10):847-867, Abril 2002.

[Benso *et al.* 1998a] A. Benso, M. Rebaudengo, L. Impagliazzo, P. Marmo, "Fault-List Collapsing for Fault-Injection Experiments", en *Proceedings 1998 Annual Reliability and Maintainability Symposium (RAMS98)*, pp. 383-388, Anaheim (California, EE.UU.), Enero 1998.

[Benso *et al.* 1998b] A. Benso, P.L. Civera, M. Rebaudengo, M. Sonza Reorda, A. Ferro, "A Hybrid Fault Injection Methodology for Real Time Systems", en *FastAbstracts 28th International Symposium on Fault-Tolerant Computing (FTCS-28)*, pp. 74-75, Munich (Alemania), Junio 1998.

[Benso *et al.* 1998c] A. Benso, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, "A Fault Injection Environment for Microprocessor-based Boards", en *Proceedings International Test Conference 1998 (ITC'98)*, pp. 768-773, Washington (D.C., EE.UU.), Octubre 1998.

[Benso *et al.* 1998d] A. Benso, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, "EXFI: A Low Cost Fault Injection System for Embedded Microprocessor-Based Boards", *ACM Transactions on Design Automation of Electronic Systems*, 3(4):626-634, Octubre 1998.

[Benso *et al.* 1999a] A. Benso, P.L. Civera, M. Rebaudengo, M. Sonza Reorda, "A Low-Cost Programmable Board for Speeding-Up Fault Injection in Microprocessor-Based Systems", en *Proceedings 1999 Annual Reliability and Maintainability Symposium (RAMS99)*, pp. 171-177, Washington (D.C., EE.UU.), Enero 1999.

[Benso *et al.* 1999b] A. Benso, M. Rebaudengo, M. Sonza Reorda, "FlexFi: A Flexible Fault Injection Environment for Microprocessor-Based Systems", en *Proceedings 18th International Conference on Computer Safety, Reliability and Security (SAFECOMP'1999)*, pp. 323-335, Toulouse (Francia), Setiembre 1999.

[Berrojo *et al.* 2002] L. Berrojo, I. González, F. Corno, M. Sonza Reorda, G. Squillero, L. Entrena, C. López, "New Techniques for Speeding-up Fault Injection Campaigns", en *Proceedings 2002 Design Automation and Test in Europe (DATE2002)*, pp. 847-852, París (Francia), Marzo 2002.

[Borel 2001] J. Borel, "Silicon Redemption: Design Automation and Product Evolution", Capítulo IVa en [Leray 2001], pp. IVa-1–IVb-34, Setiembre 2001.

[Boué *et al.* 1996] J. Boué, J. Arlat, Y. Crouzet, P. Pétillon, "Verification of Fault Tolerance by Means of Fault Injection into VHDL Simulation Models", *LAAS Report nº 96-463*, Diciembre 1996.

[Boué *et al.* 1998] J. Boué, P. Pétillon, Y. Crouzet, "MEFISTO-L: A VHDL-Based Fault Injection Tool for the Experimental Assessment of Fault Tolerance", en *Proceedings 28th International Symposium on Fault-Tolerant Computing (FTCS-28)*, pp. 168-173, Munich (Alemania), Junio 1998.

[Bouricius *et al.* 1969] W. Bouricius, W. Carter, P. Schneider, "Reliability modeling techniques for self-repairing computer systems", en *Proceedings 24th ACM National Conference*, pp. 295-309, 1969.

[Bowen y Pradhan 1991] N.S. Bowen y D.K. Pradhan, "Program fault tolerance based on memory access behaviour", en *Proceedings 21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, pp. 426-433, Montreal (Canadá), Junio 1991.

[Burgun *et al.* 1996] L. Burgun, F. Reblewski, G. Fenelon, J. Bariber, O. Lepape, "Serial Faul Emulation", en *Proceedings 1996 Design Automation Coference (DAC)*, pp. 801-806, Las Vegas (Nevada, EE.UU.), Junio 1996.

[Calvez 1993] J.P. Calvez, "Embedded Real-Time Systems", ISBN 0-471-93563-8, John Wiley & Sons, 1993.

[Campelo 1999] J.C. Campelo, "Diseño y validación de nodos de proceso tolerantes a fallos de sistemas industriales distribuidos", *Tesis doctoral*, Departamento de Informática de Sistemas y Computadores, Universidad Politécnica de Valencia, Junio 1999.

[Cardarilli *et al.* 2002] G.C. Cardarilli, F. Kaddour, A. Leandri, M. Ottavi, S. Pontarelli, R. Velazco, "Bit flip Injection in Processor-based Architectures: A Case Study", en *Proceedings 8th International On-Line Testing Workshop (IOLTW'02)*, pp. 117-127, Isla de Bendor (Francia), Julio 2002.

[Carreira *et al.* 1995] J. Carreira, H. Madeira, J.G. Silva, "Xception: a Technique for the Experimental Evaluation of Dependability in Modern Computers", en *Proceedings 5th Working Conference on Dependable Computing for Critical Applications (DCCA-5)*, pp. 135-148, Urbana-Champaign (Illinois, USA), Setiembre 1995.

[Carreira *et al.* 1998] J. Carreira, H. Madeira, J.G. Silva, "Xception: a Technique for the Experimental Evaluation of Dependability in Modern Computers", *IEEE Transactions on Software Engineering*, 24(2):125-136, Febrero 1998.

[Celeiro *et al.* 1996] F. Celeiro, L. Dias, J. Ferreira, M.B. Santos, J.P. Teixeira, "VHDL Fault Simulation for Defect-Oriented Test and Diagnosis of Digital ICs", en *Proceedings 1996 European Design Automation Conference with EURO-VHDL (EURO-DAC'96/EURO-VHDL'96)*, pp. 450-455, Ginebra (Suiza), Setiembre 1996.

[Cha *et al.* 1993] H. Cha, E.M. Rudnick, G.S. Choi, J.H. Patel, R.K. Iyer, "A Fast and Accurate Gate-Level Transient Fault Simulation Environment", en *Proceedings 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pp. 310-319, Toulouse (Francia), Junio 1993.

[Cha *et al.* 1996] H. Cha, E.M. Rudnick, J.H. Patel, R.K. Iyer, G.S. Choi, "A Gate-Level Simulation Environment for Alpha-Particle-Induced Transient Faults", *IEEE Transactions on Computes*, 45(11):1248-1256, Noviembre 1996.

[Chee y Tsang 2002] Y.H. Chee y C. Tsang, "Design of Fast and Robust State Elements", *Final Report of EE 241 Project*, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, disponible en http://www-inst.eecs.berkeley.edu/~yhchee/EE241, Mayo 2002.

[Chen 1993] M.E.Y. Chen, "Testing and evaluating fault tolerant protocols by deterministic fault injection", *Fortschritt-Beritche*, nº 260, VDI Verlag, Düsseldorf (Alemania), Julio 1993.

[Cheng *et al.* 1995] K.-T. Cheng, S.-Y. Huang, W.-J. Dai, "Fault Emulation: A New Approach to Fault Grading", en *Proceedings 1995 International Conference on Computer-Aided Design (ICCAD'95)*, pp. 681-686, 1995.

[Cheng *et al.* 1999] K.-T. Cheng, S.-Y. Huang, W.-J. Dai, "Fault Emulation: A New Methodology for Fault Grading", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(10):1487-1495, Octubre 1999.

[Choi e Iyer 1992] G.S. Choi y R.K. Iyer, "FOCUS: An experimental environment for fault sensitivity analysis", *IEEE Transactions on Computers*, 41(12):1515-1526, Diciembre 1992.

[Choi e Iyer 1993] G.S. Choi y R.K. Iyer, "Wear-out simulation environment for VLSI designs", en *Proceedings 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pp. 320-329, Toulouse (Francia), Junio 1993.

[Civera *et al.* 2001a] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante, "FPGA-Based Fault Injection Techniques for Fast Evaluation of Fault Tolerance in VLSI Circuits", en *Proceedings 11th International Conference Field Programmable Logic and Applications (FPL2001)*, pp. 493-502, Belfast (Reino Unido), Agosto 2001.

[Civera *et al.* 2001b] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante, "FPGA-Based Fault Injection for Microprocessor Systems", en *Proceedings 10th Asian Test (ATS 2001)*, pp. 304-312, Kyoto (Japón), Noviembre 2001.

[Civera *et al.* 2001c] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante, "Exploiting Circuit Emulation for Fast Hardness Evaluation", *IEEE Transactions on Nuclear Science*, 48(6):2210-2216, Diciembre 2001.

[Clark y Pradhan 1992] J.A. Clark y D.K. Pradhan, "REACT: Reliable Architecture Characterization Tool", *Technical Report TR-92-CSE-22*, University of Massachusetts, Junio 1992.

[Clark y Pradhan 1995] J.A. Clark y D.K. Pradhan, "Fault injection: A method for validating computer-system dependability", *IEEE Computer*, 28(6):47-56, Junio 1995.

[Class C 1994] "Class C application requirements-J2056/1", *SAE Handbook*, SAE Press, 1994, disponible en http://www.sae.org.

[Constantinescu 2001] C. Constantinescu, "Dependability Analysis of a Fault-Tolerant Processor", en *Proceedings 2001 Pacific Rim International Symposium on Dependable Computing (PRDC 2001)*, pp. 63-67, Seúl (Corea de Sur), Diciembre 2001.

[Constantinescu 2002] C. Constantinescu, "Impact of Deep Submicron Technology on Dependability of VLSI Circuits", en *Proceedings International 2002 Conference on Dependable Systems and Networks (DSN-2002)*, pp. 205-209, Washington (D.C., EE.UU.), Junio 2002.

[Corno *et al.* 2000] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, "RT-level Fault Simulation Techniques based on Simulation Command Scripts", en *Proceedings XV Conference on Design of Circuits and Integrated Systems (DCIS 2000)*, pp. 825-830, Montpellier (Francia), Noviembre 2000.

[Courtois *et al.* 1992] B. Courtois, M.C. Gaudel, J.C. Laprie, D. Powell, "Sûreté de Fonctionnement Informatique. Evolutions 1987-1992. Tendances et Perspectives", *Rapport rédigé à la demande de la Direction Centrale de la Qualité du CNES*, Diciembre 1992.

[Czeck y Siewiorek 1990] E.W. Czeck y D.P. Siewiorek, "Effects of transient gate-level faults on program behaviour", en *Proceedings 20th International Symposium on Fault-Tolerant Computing (FTCS-20)*, pp. 236-243, Newcastle Upon Tyne (Reino Unido), Junio 1990.

[Czeck y Siewiorek 1992] E.W. Czeck y D.P. Siewiorek, "Observations on the effects of fault manifestation as a function of workload", *IEEE Transactions on Computers*, 41(5):559-566, Mayo 1992.

[Damm 1986] A. Damm, "The effectiveness of software error detection mechanisms in real time operating systems", en *Proceedings 16th International Symposium on Fault-Tolerant Computing (FTCS-16)*, pp. 171-176, Viena (Austria), Julio 1986.

[Damm 1988] A. Damm, "Experimental Evaluation of error detection and self checking coverage of components of a distributed real-time systems", *Tesis doctoral*, Universität Wien (Austria), Octubre 1988.

[Dawson *et al.* 1996a] S. Dawson, F. Jaharnian, T. Mitton, T.-L. Tung, "Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection", en *Proceedings 26th International Symposium on Fault-Tolerant Computing (FTCS-26)*, pp. 404-414, Sendai (Japón), Junio 1996.

[Dawson *et al.* 1996b] S. Dawson, F. Jaharnian, T. Mitton, "ORCHESTRA: A Fault Injection Environment for Distributed Systems", *Technical Report CSE-TR-318-96,* University of Michigan, Electrical Engineering and Computer Science Department (EECS), 1996.

[DBench 2001] "State of the Art", *Deliverable CF1 del proyecto Dependability Benchmarking (DBench), IST-2000-25245*, Agosto 2001.

[DBench 2002] P.J. Gil *et al.*, "Fault Representativeness", *Deliverable ETIE2 of Dependability Benchmarking Project (DBench), IST-2000-25245*, Julio 2002.

[DBench 2003] "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.

[De Millo *et al.* 1987] R.A. De Millo, W.M. McCracken, R. Martin, J.F. Passafiume, "Software Testing and Evaluation", Benjamin/Cummings Publ. Company, 1987.

[DeLong *et al.* 1994] T.A. DeLong, B.W. Johnson, J.A. Profeta III, D. Bozzolo, "A Novel Fault Injection Technique for Behavioral-Level Modeling using VHDL", en *Proceedings 1994 VHDL International Users' Forum - Fall Conference (VIUF FALL 94)*, pp. 9.13-9.21, Mc Lean (Virginia, EE.UU.), Noviembre 1994.

[DeLong *et al.* 1996] T.A. DeLong, B.W. Johnson, J.A. Profeta III, "A Fault Injection Technique for VHDL Behavioral-Level Models", *IEEE Design & Test of Computers*, 13(4):24-33, Invierno 1996.

[Dewey y Geus 1992] A. Dewey A y A.J.D. Geus, "VHDL: Toward a Unified View of Design", *IEEE Design and Test of Computers*, 9(2):8-17, Abril/Junio 1992.

[Dugan y Trivedi 1989] J.B. Dugan y K.S. Trivedi, "Coverage modelling for dependability analysis of fault-tolerance systems", *IEEE Transactions on Computers*, 38(6):775-787, Junio 1989.

[Dupuy *et al.* 1990] A. Dupuy, J. Schwartz, Y. Yemini, D. Bacon, "NEST: A Network Simulation and Prototyping Testbed", *Communications of the ACM*, 33(10):64-74, Octubre 1990.

[Echtle y Chen 1991] K. Echtle e Y. Chen, "Evaluation of deterministic fault injection for fault tolerant protocol testing", en *Proceedings 21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, pp. 418-425, Montreal (Canadá), Junio 1991.

[Echtle y Leu 1992] K. Echtle y M. Leu, "The EFA fault injector for fault tolerant distributed system testing", en *Proceedings IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pp. 28-35, Amherst (EE.UU.), Julio 1992.

[Fabre *et al.* 1999] J.C. Fabre, F. Sallés, M. Rodríguez, J. Arlat, "Assessment of COTS microkernel by fault injection", en *Proceedings 7th IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-7)*, pp. 19-38, San Jose (EE.UU.), Enero 1999.

[Fang y McVittie 1993] S. Fang y J. McVittie, "A Mechanism for Gate Oxide Damage in Nonuniform Plasmas", en *Proceedings 31st International Reliability Physics Symposium (IRPS '93)*, pp. 13-17, Atlanta (Georgia, EE.UU.), Marzo 1993.

[FAST 2001] "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.

[Favalli *et al.* 1991] M. Favalli, P. Olivo, and B. Riccò, "Fault Simulation for General FCMOS ICs", *Journal of Electronic Testing: Theory and Applications*, 2(2):181-190, Junio 1991.

[Finelli 1987] G.B. Finelli, "Characterisation of fault recovery through fault injection on FTMP", *IEEE Transactions on Reliability*, 36(2):164-170, Junio 1987.

[FIT 2002a] "Consolidated Status Report – Measurements", *Consolidated Status Report of Faul Injection for TTA Project (FIT), IST-1999-10748*, Febrero 2002.

[FIT 2002b] "Deliverable 5.1 – 5.5: Combined Report", *Deliverable 5 of Faul Injection for TTA Project (FIT), IST-1999-10748*, Setiembre 2002.

[FIT 2002c] "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.

[Folkesson *et al.* 1998] P. Folkesson, S. Svensson, J. Karlsson, "A comparison of simulation based and scan chain implemented fault injection", en *Proceedings 28th International Symposium on Fault-Tolerant Computing (FTCS-28)*, pp. 284-293, Munich (Alemania), Junio 1998.

[Folkesson 1999] P. Folkesson, "Assessment and Comparison of Physical Fault Injection Techniques", *Thesis for the degree of Doctor of Philosophy*, Department of Computer Engineering, Chalmers University of Technology, Göteborg (Suecia), 1999.

[Freeman 1996] L.B. Freeman, "Critical Charge Calculations for a Bipolar SRAM Array", *IBM Journal of Research and Development*, 40(1), Enero 1996.

[Fuchs 1996] E. Fuchs, "An Evaluation of the Error Detection Mechanisms in MARS Using Software-Implemented Fault Injection", en *Proceedings 2nd European Dependable Computing Conference (EDCC-2)*, pp. 73-90, Taormina (Italia), Octubre 1996.

[Gaisler 1997] J. Gaisler, "Evaluation of a 32-bit Microprocessor with Built-in Concurrent Error Detection", en *Proceedings 27th International Symposium on Fault-Tolerant Computing (FTCS-27)*, pp. 42-47, Seattle (Washington, EE.UU.), Junio 1997.

[Gaisler 2002] J. Gaisler, "A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8 Architecture", en *Proceedings 2002 International Conference on Dependable Systems and Networks (DSN 2002)*, pp. 409-415, Bethesda (Maryland, EE.UU.), Junio 2002.

[Galiay *et al.* 1980] J. Galiay, Y. Crouzet, M. Vergniault, "Physical Versus Logical Fault Models MOS LSI Circuits: Impact on Their Testability", *IEEE Transactions on Computer*s, 29(6):527-531, Junio 1980.

[Gautier y Leroy 2001] J. Gautier y J.-L. Leray, "Silicon Redemption: Technology Roadmap", Capítulo IVb en [Leray 2001], pp. IVb-1–IVb-36, Setiembre 2001.

[Ghate 1981] P.B. Ghate, "Aluminum Alloy Metallization for Integrated Circuits", *Thin Solid Films*, 83(2):195-205, Setiembre 1981.

[Ghosh y Chakraborty 1991] S. Ghosh y T.J. Chakraborty, "On behavior fault modeling for digital design", *Journal of Electronic Testing: Theory and Applications*, (2):135-151, 1991.

[Ghosh *et al.* 1995a] A.K. Ghosh, B.W. Johnson, J.A. Profeta III, "System-Level Modeling in the ADEPT Environment of a Distributed Computer System for Real-Time Applications", en *Proceedings 1995 International Computer Performance and Dependability Symposium (IPDS-95)*, pp. 194-203, Erlangen (Alemania), Abril 1995.

[Ghosh *et al.* 1995b] A.K. Ghosh, T.A. DeLong, B.W. Johnson, "Fault Injection in the Design Process Using VHDL", en *Proceedings 1995 VHDL Users Group/VHDL International Users' Forum - Fall Conference (VUG/VIUF FALL 95)*, Boston (Massachusetts, EE.UU.), Octubre 1995.

[Gil 1992] P.J. Gil, "Sistema Tolerante a Fallos con Procesador de Guardia: Validación mediante Inyección Física de Fallos", *Tesis doctoral*, Departamento de Ingeniería de Sistemas, Computadores y Automática (DISCA), Universidad Politécnica de Valencia, Septiembre 1992.

[Gil 1996] P.J. Gil, "Garantía de funcionamiento: Conceptos básicos y terminología", *Informe interno*, Departamento de Ingeniería de Sistemas, Computadores y Automática (DISCA), Universidad Politécnica de Valencia, 1996.

[Gil 1999] D. Gil, "Validación de Sistemas Tolerantes a Fallos mediante inyección de fallos en modelos VHDL", *Tesis Doctoral*, Departamento de Informática de Sistemas y Computadores (DISCA), Universidad Politécnica de Valencia, 1999.

[Gil *et al.* 1997a] P.J. Gil, J.C. Baraza, D. Gil, J.J. Serrano, "High speed fault injector for its application in the safety validation of industrial machinery", en *8th European Workshop on Dependable Computing (EWDC-8): "Experimental validation of dependable systems"*, Göteborg (Suecia), Abril 1997.

Gil *et al.* 1997b] D. Gil, J.C. Baraza, J.V. Busquets, L. Lemus, J. Albaladejo, P.J. Gil, "Inyección de fallos mediante simulación de modelos VHDL. Aplicación al estudio de un sistema computador sencillo", en *Actas del Seminario Anual de Automática, Electrónica Industrial e Instrumentación 1997 (SAAEI'97)*, Valencia, pp. 758-763, Setiembre 1997.

[Gil *et al.* 1997c] D. Gil, J.C. Baraza, J.V. Busquets, P.J. Gil, "Fault Injection with Simulation in VHDL Models and Its Application to a Simple Microcomputer System", en *Proceedings 5th International Conference on Advanced Computing (ADCOMP 97)*, Chennai (India), pp. 466-474, Diciembre 1997.

[Gil *et al.* 1998a] D. Gil, J.V. Busquets, J.C. Baraza, P.J. Gil, "A Fault Injection Tool for VHDL Models", en *Digest of Abstracts 28th International Symposium on Fault-Tolerant Computing (FTCS-28)*, Munich (Alemania), pp. 72-73, Junio 1998.

[Gil *et al.* 1998b] D. Gil, J.C. Baraza, J.V. Busquets, P.J. Gil, "Fault Injection into VHDL Models: Analysis of the Error Syndrome of a Microcomputer System", en *Proceedings 24th EUROMICRO Conference*, Västerås (Suecia), 1:418-425, Agosto 1998.

[Gil *et al.* 1998c] D. Gil, J.V. Busquets, J.C. Baraza, P.J. Gil, *Using VHDL in the Techniques of Fault Injection based on Simulation*. Proc. XIII Design of Circuits and Integrated Systems Conference (DCIS-98), Madrid, pp. 174-180, Noviembre 1998.

[Gil *et al.* 1998d] D. Gil, J.C. Baraza, J.V. Busquets, P.J. Gil, "Inyección de Fallos Mediante Simulación de Modelos VHDL. Aplicación al estudio de un Sistema Computador Sencillo", *Información Tecnológica*, ISSN 0716-8756, La Serena (Chile), 9(6):111-118, Noviembre-Diciembre 1998.

[Gil *et al.* 1999] D. Gil, R.J. Martínez, J.V. Busquets, J.C. Baraza, P.J. Gil, "Fault Injection into VHDL Models: Experimental Validation of a Fault Tolerant Microcomputer System", en *Proceedings 3rd European Dependable Computing Conference (EDCC-3)*, Praga (República Checa), pp. 191-208, Setiembre 1999.

[Gil *et al.* 2000] D. Gil, J. Gracia, J.C. Baraza, P.J. Gil, "A Study of the Effects of Transient Fault Injection into the VHDL Model of a Fault Tolerant Microcomputer System", en *Proceedings 6th IEEE International On-Line Testing Workshop (IOLTW 2000)*, Palma de Mallorca, pp. 73-79, Julio 2000.

[Gil *et al.* 2003a] D. Gil, J. Gracia, J.C. Baraza, P.J. Gil, "Study, Comparison and Application of different VHDL-Based Fault Injection Techniques for the Experimental Validation of a Fault-Tolerant System", *Microelectronics Journal*, 34(1):41-51, Enero 2003.

[Gil *et al.* 2003b] D. Gil, J.C. Baraza, J. Gracia, P.J. Gil, "VHDL Simulation-Based Fault Injection Techniques", en *Fault Injection Techniques and Tools for VLSI Reliability Evaluation*, pendiente de publicación, 2003.

[Globetrotter 1998] Globetrotter Software, Inc., "FLEXlm End User Manual", Disponible en http://www.globetrotter.com/endu-ser.pdf, Globetrotter Software, Inc., 1998.

[Goswami e Iyer 1992] K.K. Goswami y R.K. Iyer, "DEPEND: A Simulation-Based Environment for System Level Dependability Analysis", *Technical Report CRHC 92-11*, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana (Illinois, EE.UU.), Junio 1992.

[Goswami e Iyer 1993] K.K. Goswami y R.K. Iyer, "Simulation of software behaviour under hardware faults", en *Proceedings 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pp. 340-347, Toulouse (Francia), Junio 1993.

[Goswami *et al.* 1997] K.K. Goswami, R.K. Iyer, L. Young, "DEPEND: A Simulation-Based Environment for System Level Dependability Analysis", *IEEE Transactions on Computers*, 46(1):60-74, Enero 1997.

[Gracia *et al.* 2000] J. Gracia, D. Gil, J.C. Baraza, P.J. Gil, "Application of Different VHDL-Based Fault Injection Techniques to the Validation of a Fault Tolerant Microcomputer System", en *Workshops and Abstracts of the International Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8)*, Nueva York (EE.UU.), pp. B-54–B-55, Junio 2000.

[Gracia *et al.* 2001a] J. Gracia, J.C. Baraza, D. Gil, P.J. Gil, "A Study of the Experimental Validation of Fault-Tolerant Systems Using Different VHDL-Based Fault Injection Techniques", en *Proceedings 7th IEEE International On-Line Testing Workshop (IOLTW 2001)*, Taormina (Italia), pág. 140, Junio 2001.

[Gracia *et al.* 2001b] J. Gracia, J.C. Baraza, D. Gil, P.J. Gil, "Comparison and Application of Different VHDL-Based Fault Injection Techniques", en *Proceedings 2001 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2001)*, San Francisco (EE.UU.), pp. 233-241, Octubre 2001.

[Gracia *et al.* 2002a] J. Gracia, S. Blanc, J.C. Baraza, D. Gil, P.J. Gil, "VFIT: Una herramienta automática para la inyección de fallos en VHDL", en *Actas Seminario Anual de Automática, Electrónica Industrial e Instrumentación (SAAEI 2002)*, Alcalá de Henares (Madrid), pp. II-289–II-292, Setiembre 2002.

[Gracia *et al.* 2002b] J. Gracia, D. Gil, L.G. Lemus, P.J. Gil, "Studying Hardware Fault Representativeness with VHDL Models", en *Proceedings XVII Design of Circuits and Integrated Systems Conference (DCIS'02)*, Santander, pp. 33-38, Noviembre 2002.

[Gracia *et al.* 2002c] J. Gracia, D. Gil, J.C. Baraza, P.J. Gil, "Using VHDL-Based Fault Injection to exercise Error Detection Mechanisms in the Time-Triggered Architecture", en *Proceedings 2002 Pacific Rim International Symposium on Dependable Computing (PRDC 2002)*, Tsukuba (Japón), pp. 316-320, Diciembre 2002.

[Gracia *et al.* 2003] J. Gracia, J.C. Baraza, D. Gil, P.J. Gil, "Early Diagnosis of Hard Real-Time Fault-Tolerant Embedded Systems", en *Proceedings 6th International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2003)*, Poznań (Polonia), pp.157-164, Abril 2003.

[Gunneflo *et al.* 1989] U. Gunneflo, J. Karlsson, J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation", en *Proceedings 19th International Symposium on Fault-Tolerant Computing (FTCS-19)*, pp. 218-227, Chicago (Illinois, EE.UU.), Junio 1989.

[Gunneflo 1990] U. Gunneflo, "The effects of power supply disturbances on the MC6809E microprocessor", *Technical Report 89*, Department of Computer Engineering, Chalmers University of Technology, Göteborg (Suecia), 1990.

[Güthoff y Sieh 1995] J. Güthoff y V. Sieh, "Combining Software-Implemented and Simulation-Based Fault Injection into a Single Fault Injection Method", en *Proceedings 25th International Symposium on Fault-Tolerant Computing (FTCS-25)*, pp. 196-206, Pasadena (California, EE.UU.), Junio 1995.

[Han *et al.* 1994] S. Han, H.A. Rosenberg, K.G. Shin, "DOCTOR: An integrated software fault injection environment", en *Proceedings 3rd IEEE International Workshop on integrating error models with fault injection*, Annapolis (EE.UU.), Abril 1994.

[Hawkins 2000] C. Hawkins, "CMOS IC Failure Mechanism and Defect Based Testing", en *2nd summer course on selected microelectronic design & test topics*, Palma de Mallorca, Julio 2000.

[Hazucha y Svensson 2000] P. Hazucha y C. Svensson, "Impact of CMOS Technology Scaling on the Atmospheric Neutron Soft Error Rate", *IEEE Transactions on Nuclear Science*, 47(6):2586-2594, Diciembre 2000.

[Holbert 2003] K.E. Holbert, "Single Event Effects", en *EEE 460 Nuclear Concepts for the 21st Century*, Curso impartido por el profesor Holbert, Electrical Engineering Departament, Arizona State University, disponible en http://www.asu.edu/~holbert/eee460/see.html.

[Hong *et al.* 1996] J.-H. Hong, S.-A. Hwang, C.-W. Wu, "An FPGA-Based Hardware Emulator for Fast Fault Emulation", en *Proceedings 1996 Midwest Symposium on Circuit and Systems*, Ames (Iowa, EE.UU.), Agosto 1996.

[Höxer *et al.* 2002] H.-J. Höxer, K. Buchacker, V. Sieh, "UMLinux – A Tool for Testing a Linux System's Fault Tolerance", en *Proceedings LinuxTag 2002*, Karlsruhe (Alemania), Junio 2002.

[Hsueh *et al.* 1997] M. Sueh, T. Tsai, R.K. Iyer, "Fault Injection Techniques and Tools", *IEEE Computer*, 30(4):75-82, Abril 1997.

[Hu y Lu 1999] C. Hu y Q. Lu, "A Unified Gate Oxide Reliability Model", en *Proceedings 39th International Reliability Physics Symposium (IRPS '99)*, pp. 47-52, San Diego (California, EE.UU.), Marzo 1999.

[Hwang *et al.* 1998] S.-A. Hwang, J.-H. Hong, C.-W. Wu, "Sequential Circuit Fault Simulation Using Logic Emulation", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(8):724-736, Agosto 1998.

[IEEE 1988] Institute of Electric and Electronic Engineers (IEEE), "IEEE Standard VHDL Language Reference Manual", *ANSI/IEEE Std 1076-1987*, 1988.

[IEEE 1994] Institute of Electric and Electronic Engineers (IEEE), "IEEE Standard VHDL Language Reference Manual", *ANSI/IEEE Std 1076-1993 (Revision of IEEE Std 1076-1987)*, 1994.

[Iyer y Rosseti 1986] R.K. Iyer y D. Rosseti, "A measurement-based model for workload dependence of CPU errors", *IEEE Transactions on Computers*, 35(6):511-519, Junio 1986.

[Iyer 1995] R.K. Iyer, "Experimental Evaluation", en *Proceedings 25th International Symposium on Fault-Tolerant Computing (FTCS-25) – Special Issue*, pp. 115-132, Pasadena (California, EE.UU.), Junio 1995.

[Jagannath y Rai 1995] A. Jagannath y S. Rai, "Impact of hardware and software faults on ARQ schemes – An experimental study", en *Proceedings Annual Reliability and Maintainability Symposium*, pp. 479-485, Washington (EE.UU.), Enero 1995.

[Jenn 1994] E. Jenn, "Sur la validation des systémes tolérant les fautes: injection de fautes dans des modéles de simulation VHDL", *Thèse*, Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS (LAAS), LAAS Report nº 94-361, 1994.

[Jenn y Arlat 1992] E. Jenn y J. Arlat, "Mise en Oeuvre de l'Injection de Fautes en VHDL", *LAAS Report nº 92-66*, Julio 1992.

[Jenn *et al.* 1993a] E. Jenn, J. Arlat, Y. Crouzet, "Fault Injection into VHDL Models for the Validation of Fault-Tolerant Systems", *LAAS Report nº 93-030*, Febrero 1993.

[Jenn *et al.* 1993b] E. Jenn, J. Arlat, M. Rimén, J. Ohlsson, J. Karlsson, "Fault Injection into VHDL Models: The MEFISTO Tool", *LAAS Report nº 93-460*, Diciembre 1993.

[Jenn *et al.* 1994] E. Jenn, J. Arlat, M. Rimén, J. Ohlsson, J. Karlsson, "Fault injection into VHDL models: the MEFISTO tool", en *Proceedings 24th International Symposium on Fault-Tolerant Computing (FTCS-24)*, pp. 356-363, Austin (Texas, EE.UU.), Junio 1994.

[Jones 1987] R.E. Jones, "Line Width Dependence of Stresses in Aluminium Interconnect", en *Proceedings 25th International Reliability Physics Symposium (IRPS* '87*)*, pp. 9-14, San Diego (California, EE.UU.), Abril 1987.

[Juhnke y Klar 1995] T. Juhnke y H Klar, "Calculation of the Soft Error Rate of Submicron CMOS Logic Circuits", *IEEE Journal of Solid-State Circuits*, 30:830–834, Julio 1995.

[Kanawati *et al.* 1992] G.A. Kanawati, N.A. Kanawati, J.A. Abraham, "FERRARI: A tool for the validation of system dependability properties", en *Proceedings 22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, pp. 336-344, Boston (Massachusetts, EE.UU.), Julio 1992.

[Kanawati *et al.* 1995] G.A. Kanawati, N.A. Kanawati, J.A. Abraham, "FERRARI: A flexible software based fault and error injection system", *IEEE Transactions on Computers*, 44(2):248-260, Febrero 1995.

[Kanoun 1989] K. Kanoun, "Croissance de la Sûréte de fonctionement des logiciels. Caracterisation – Modelisation – Evaluation", *Thése présentée a L'Institut National Polytechnique de Toulouse*, Setiembre 1989.

[Kao *et al.* 1993] W. Kao. R.K. Iyer, D. Tang, "FINE: a fault injection and monitoring environment for tracing UNIX system behaviour under faults", *IEEE Transactions on Software Engineering*, 19(11):1105-1118, Noviembre 1993.

[Kao e Iyer 1994] W. Kao y R.K. Iyer, "DEFINE: a distributed fault injection and monitoring environment", en *Workshop on fault tolerant parallel and distributed systems*, Junio 1994.

[Karlsson 1990] J. Karlsson, "Transient fault effects in the MC6809E 8-bit microprocessor: A comparison of results of physical and simulated fault injection experiments", *Technical Report 96*, Department of Computer Engineering, Chalmers University of Technology, Göteborg (Suecia), 1990.

[Karlsson *et al.* 1989] J. Karlsson, U. Gunneflo, J. Torin, "Use of heavy-ion radiation from 252-californium for fault injection experiments", en *Proceedings 1st International Working Conference on Dependable Computing for Critical Applications (DCCA-1)*, Santa Barbara (EE.UU.), Agosto 1989.

[Karlsson *et al.* 1995] J. Karlsson, P. Folkensson, J. Arlat, Y. Crouzet, G. Leber, "Integration and comparison of three physical fault injection techniques", en *Predictably Dependable Computing Systems*, Capítulo 5, pp. 309-329, Springer-Verlag, 1995.

[Kilty 1995] P. Kilty, "VHDL/VITAL Fault Simulation", en *Proceedings 1995 VHDL Users Group/VHDL International Users' Forum - Fall Conference (VUG/VIUF FALL 95)*, Boston (Massachusetts, EE.UU.), Octubre 1995.

[King *et al.* 1989] R. King, C. van Schaick, J. Lusk, "Electrical Overstresses of Non-Encapsulated Bond Wires", en *Proceedings 27th International Reliability Physics Symposium (IRPS '89)*, pp.141-151, Phoenix (Arizona, EE.UU.), Abril 1989.

[Kopetz 1998] H. Kopetz, "The Time-Triggered Architecture", en *Proceedings 1998 IFIP International Workshop on Dependable Computing and Its Applications (DCIA 98)*, pp. 203-214, Johannesburgo (República de Sudáfrica), Enero 1998.

[Kopetz 1999] H. Kopetz, "TTP/C Protocol", disponible en http://www.ttpforum.org, 1999.

[Kumar *et al.* 1994] S. Kumar, R.H. Klence, J.H. Aylor, B.W. Johnson, R.D. Williams, R. Waxman, "ADEPT: A Unified System Level Modeling Design Environment", en *Proceedings 1st Rapid Prototyping of Application Specific Signal Processors Conference (RASSP'94)*, pp. 114-123, Arlington (Virginia, EE.UU.), Agosto 1994.

[Lala 1983] J.H. Lala, "Fault detection, isolation, and reconfiguration in FTMP: Methods and Experimental Results", en Procedings 5th AIAA/IEEE Digital Avionics Systems Conference, pp. 21.3.1-21.3.9, 1983.

[Laprie 1985] J.C. Laprie, "Dependable Computing and Fault-Tolerance: Concepts and Terminology", en *Proceedings 15th IEEE International Symposium on Fault-Tolerant Computing (FTCS-15)*, pp. 2-11, Ann Arbor (Michigan, EE.UU.), Junio 1985.

[Laprie 1995] J.C. Laprie, "Dependable Computing: Concepts, Limits, Challenges", en *Proceedings 25th IEEE International Symposium on Fault-Tolerant Computing (FTCS-25)*, pp. 42-54, Pasadena (California, EE.UU.), Junio 1995.

[Lee *et al.* 2001] W.H. Lee, D.-K. Lee, Y.-M. Park, K.-S. Kim, K.-O. Ahn, K.-D. Suh, "Data Retention Failure in NOR Flash Memory Cells", en *Proceedings 39th International Reliability Physics Symposium (IRPS '01)*, pp. 57-61, Orlando (Florida, EE.UU.), Abril-Mayo 2001.

[LEON-2 2003] Gaisler Research, "LEON2 processor", disponible en http://www.gaisler.com/leon.html, 2003.

[Leroy 2001] J.-L. Leray ed., "Earth and Space Single-Events in Present and Future Electronics", *RADECS 2001 Short Course*, ISBN 2-911798-02-3, Setiembre 2001.

[Leveugle 2000] R. Leveugle, "Fault Injection in VHDL Descriptions and Emulation", en *Proceedings 2000 International Symposium on Defect and Fault Tolerance (DFT'00)*, pp. 414-420, Yamanashi (Japón), Octubre 2000.

[Leveugle 2001] R. Leveugle, "A Low-Cost Hardware Approach to Dependability Validation of IPs", en *Proceedings 2001 International Symposium on Defect and Fault Tolerance (DFT'01)*, pp. 242-249, San Francisco (California, EE.UU.), Octubre 2001.

[Leveugle y Hadjiat 2000] R. Leveugle y K. Hadjiat, "Optimized Generation of VHDL Mutants for Injection of Transition Errors" en *Proceedings 13th Symposium on Integrated Circuits and Systems Design (SBCCI'00)*, pp. 243-248, Manaus (Brasil), Setiembre 2000.

[Li y Wu 1995] Y.-L. Li y C.-W. Wu, "Cellular Automata for Efficient Parallel Logic and Fault Simulation", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(6):740-749, Junio 1995.

[Li *et al.* 1997] Y.-L. Li, Y.-C. Lai, C.-W. Wu, "VLSI Design of a Cellular-Automata Based Logic and Fault Simulator", en *Proceedings 1997 National Science Council Part A: Physical Science and Engineering*, 21:189-199, Taipei (Taiwan, China), Mayo 1997.

[Lidén *et al.* 1994] P. Lidén, P. Dahlgren, R. Johansson, J. Karlsson, "On Latching Probability of Particle Induced Transients in Combinatorial Networks", en *Proceedings 24th International Symposium on Fault Tolerant Computer Systems (FTCS-24)*, pp. 821-824, Junio 1994.

[Lima *et al.* 2001] F. Lima, S. Rezgui, L. Carro, R. Velazco, R. Reis, "On the Use of VHDL Simulation and Emulation to Derive Error Rates", en *Proceedings 6th European Conference on Radiation and its Effects on Components and Systems (RADECS 2001)*, Grenoble (Francia), Setiembre 2001.

[Lin *et al.* 1996] H.C. Lin, C.H.Peng, C.H.Chien, T.F.Chang, T.Y.Huang, C.Y.Chang, "Plasma Charging Induced Gate Oxide Damage During Metal Etching and Ashing", en *Proceedings 1st International Symposium on Plasma Process-Induced Damage (P2ID)*, pp. 113-116, Santa Clara (California, EE.UU.), Mayo 1996.

[LIS 1996] Laboratoire d'Ingenierie de la Sûreté de fonctionnement (LIS), "Guide de la sûreté de fonctionnement", Cépaduès – Éditions, 1996.

[Lomelino e Iyer 1986] D. Lomelino y R.K. Iyer, "Error Propagation in a Digital Avionic Processor – A Simulation-Based Study", en *Proceedings 1986 Real-Time Systems Symposium (RTSS)*, pp. 218-225, New Orleans (Louisiana, EE.UU), Diciembre 1986.

[Lovric y Echtle 1993] T. Lovric y K. Echtle, "ProFI: Processor fault injection for dependability validation", en *Proceedings International Workshop on Fault and Error Detection for Dependability Validation of Computer Systems*, Göteborg (Suecia), Junio 1993.

[Madeira *et al.* 1994] H. Madeira, M. Rela, F. Moreira, J.G Silva, "RIFLE: A general purpose pin-level fault injector", en *Proceedings 1st European Dependable Computing Conference (EDCC-1)*, pp 199-216, Berlin (Alemania), Octubre 1994.

[Majzik y Bondavalli 1998] I. Majzik y A. Bondavalli, "Dependability Analysis in the HIDE Framework", *HIDE Document*, Junio 1998.

[Martinet 1992] B. Martinet, "Contribution à l'Evaluation de l'Efficacité du Test Fonctionnel de Microprocesseurs", *Thèse de Doctorat de troisième cycle*, Institut National Politechnique de Grenoble, Noviembre 1992.

[Martínez *et al.* 1999] R.J. Martínez, P.J. Gil, G. Martín, C. Pérez, J.J. Serrano, "Experimental validation of high speed fault tolerant systems using physical fault injection", en *Proceedings 7th IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-7)*, pp. 233-250, San Jose (EE.UU.), Enero 1999.

[Mavis y Eaton 2000] D.G. Mavis y P.H. Eaton, "SEU and SET Mitigation Techniques for FPGA Circuit and Configuration Bit Storage Design", en *3rd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD'2000)*, Laurel (Maryland, EE.UU.), Setiembre 2000.

[Maxion y Olszewski 1993] R.A. Maxion y R.T. Olszewski, "Detection and discrimination of injected network faults", en *Proceedings 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pp. 198-207, Toulouse (Francia), Junio 1993.

[MC8051 2002] Oregano Systems, "MC8051 IP Core. Synthesizeable Microcontroller VHDL IP-Core. User Guide", disponible en http://oregano.at/ip/mc8051/mc8051_overview.pdf, Enero 2002.

[McCluskey 1986] E.J. Mc Cluskey. "Logic Design Principles. With emphasis on testable semicustom circuits", Prentice-Hall, ISBN 0-13-5397-847, 1986.

[McPherson 1994] J. McPherson, "Reliability/Processing Challenges for LSI Metallization", en *Tutorial Notes 32nd International Reliability Physics Symposium (IRPS '94)*, pp. 8.1-8.48, San Jose (California, EE.UU.), Abril 1994.

[McVittie 1996] J. McVittie, "Plasma Charging Damage: An Overview", en *Proceedings 1st International Symposium on Plasma Process-Induced Damage (P2ID)*, pp. 7-20, Santa Clara (California, EE.UU.), Mayo 1996.

[Meyer 1986] P.L. Meyer, "Probabilidad y aplicaciones estadísticas", ISBN 0-201-51877-5, Addison-Wesley Iberoamericana, 1986.

[Meyer y Wei 1988] J.F. Meyer y L. Wei, "Analysis of workload influence on dependability", en *Proceedings 18th International Symposium on Fault-Tolerant Computing (FTCS-18)*, pp. 84-89, Tokio (Japón), Junio 1988.

[Miczo 1990] A. Miczo, "VHDL as a Modeling-for-Testability Tool", en *Proceedings 35th Computer Society International Conference (COMPCON'90)*, pp.403-409, San Francisco (California, EE.UU.), Febrero-Marzo 1990.

[MIPS 2001] MIPS Technologies, "MIPS32 Architecture for Programmers, volume I: Introduction to the MIPS32 Architecture", 2001.

[Miremadi *et al.* 1992] G. Miremadi, J. Karlsson, U. Gunneflo, J. Torin, "Two software techniques for on line error detection", en *Proceedings 22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, pp. 328-335, Boston (Massachusetts, EE.UU.), Julio 1992.

[Miremadi y Torin 1995] G. Miremadi y J. Torin, "Evaluating processor-behaviour and three error detection mechanisms using physical fault-injection", *IEEE Transactions on Reliability*, 44(3):441-454, Setiembre 1995.

[ModelTech 1997] Model Technology, "V-System/PLUS PC User´s Manual. Version 4.6", 1997.

[ModelTech 2001a] Model Technology, "ModelSim SE User's Manual. Version 5.5e", 2001.

[ModelTech 2001b] Model Technology, "ModelSim SE Command Reference. Version 5.5e", 2001.

[Nexus 1999] The Nexus Forum™ Standard for a Global Embedded Processor Debug Interface. IEEE-ISTO 5001-1999, en http://www.ieee-isto.org/Nexus5001, 1999.

[Oates 1993] A. Oates, "Electromigration in Stress-Voided Al Alloy Metallizations for Submicron IC Technologies", en *Proceedings 31st International Reliability Physics Symposium (IRPS '93)*, pp. 297-303, Atlanta (Georgia, EE.UU.), Marzo 1993.

[Oates 1994] A. Oates, "Thin Film Electromigration: Al Alloy Metallizations for Submicron IC Technologies", en *Tutorial Notes 32nd International Reliability Physics Symposium (IRPS '94)*, pp. 2.1-2.23, San Jose (California, EE.UU.), Abril 1994.

[Ogawa *et al.* 2001] E.T. Ogawa, K.-D. Lee, H. Matsuhashi, K.-S. Ko, P.R. Justison, A.N. Ramamurthi, A.J. Bierwag, P.S. Ho, "Statistics of Electromigration Early Failures in Cu/oxide Dual-Damascene Interconnects", en *Proceedings 39th International Reliability Physics Symposium (IRPS '01)*, pp. 341-350, Orlando (Florida, EE.UU.), Abril-Mayo 2001.

[Ohlsson *et al.* 1992] J. Ohlsson, M. Rimén, U. Gunneflo, "A study of the effect of transient fault injection into a 32-bit RISC with built-in watchdog", en *Proceedings 22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, pp. 316-325, Boston (Massachusetts, EE.UU.), Julio 1992.

[Pagaduan *et al.* 2001] F.E. Pagaduan, J.K. Lee, V. Vedagarbha, K. Lui, M.J. Hart, D. Gitlin, T. Takaso, S. Kamiyama, K. Nakayama, "The Effects of Plasma Induced Damage on Transistor Degradation and the Relationship to Field Programmable Gate Array Performance", en *Proceedings 39th International Reliability Physics Symposium (IRPS '01)*, pp. 315-319, Orlando (Florida, EE.UU.), Abril-Mayo 2001.

[Parrotta *et al.* 2000] B. Parrotta, M. Rebaudengo, M. Sonza Reorda, M. Violante, "New Techniques for Accelerating Fault Injection in VHDL Descriptions", en *Proceedings 6th International On-Line Test Workshop (IOLTW'00)*, pp. 61-66, Palma de Mallorca, Julio 2000.

[PIC16C5X 1998] "Synthesizable VHDL description of PIC16C5X", disponible en http://tech-www.informatik.uni-hamburg.de/vhdl/models/pic/pic1.tar.gz.

[PIC16C5X 2003] "PIC16C5X Datasheets", disponible en http://www.microchip.com/download/lit/pline/picmicro/families/16c5x/30453d.pdf.

[Powell 1988] Delta_4 Consortium, "Delta_4: Overall System Specification", D. Powell ed., ISBN 2-907801-00-7, Diciembre 1988.

[Powell *et al.* 1999] D. Powell *et al.*, "GUARDS: A Generic Upgradable Architecture for Real-Time Dependable Systems", *IEEE Transactions on Parallel and Distributed Systems*, 10(6):580-599, Junio 1999.

[Powell 2001] "A Generic Fault-Tolerant Architecture for Real-Time Dependable Systems", ISBN 0-7923-7295-6, D. Powell, Ed., Kluwer Academic Publishers, 2001.

[Pradhan 1986] D.K. Pradhan, "Fault-Tolerant Computing Theory and Techniques", ISBN 0-13308-222-9, Prentice-Hall, 1986.

[Pradhan 1996] D.K. Pradhan, "Fault-Tolerant Computer System Design", ISBN 0-13-057887-8, Prentice-Hall, 1996.

[Proteus 1996] Proteus Corporation, "Probestar DVT-100 Application Note", Marzo 1996.

[Pucknell y Eshraghian 1994] D.A. Pucknell y K. Eshraghian, "Basic VLSI Design", Prentice Hall, 1994.

[Randell *et al.* 1995] "Predictably Dependable Computing Systems", ISBN 3-540-59334-9, B. Randell, J.C. Laprie, H. Kopetz, B. Littlewood, Eds., Springer-Verlag, 1995.

[Rangan *et al.* 1999] S. Rangan, S. Krishnan, A. Amerasekara, S. Aur, S. Ashok, "A Model for Channel Hot Carrier Reliability Degradation Due to Plasma Damage in MOS Devices", en *Proceedings 37th International Reliability Physics Symposium (IRPS '99)*, pp. 370-374, San Diego (California, EE.UU.), Marzo 1999.

[Reisinger y Steininger 1994] J. Reisinger y A. Steininger, "The design of a fail-silent processing node for MARS", *Distributed Systems Engineering Journal*, 1994.

[Riesgo y Uceda 1996] T. Riesgo y J. Uceda, "A Fault Model for VHDL Descriptions at the Register Transfer Level", en *Proceedings 1996 European Design Automation Conference with EURO-VHDL (EURO-DAC'96/EURO-VHDL'96)*, pp. 462-467, Ginebra (Suiza), Setiembre 1996.

[Rimén y Ohlsson 1993] M. Rimén y J. Ohlsson, "A study of the error behaviour of a 32 bit RISC subjected to simulated transient fault injection", *PFCS2 Report*, pp. 445-460, Setiembre 1993.

[Rimén *et al.* 1992] M. Rimén, J. Ohlsson, J. Karlsson, E. Jenn, J. Arlat, "Validation of fault tolerance by fault injection in VHDL models", *LAAS Report nº 92-469*, Diciembre 1992.

[Rodder *et al.* 1995] M. Rodder, S. Aur, C. Chen, "A Scaled 1.8V, 0.18µm Gate Lenght CMOS Technology: Device Design and Reliability Considerations", en *Technical Digest International Electron Devices Meeting (IEDM)*, pp. 415-418, Washington (D.C., EE.UU.), Diciembre 1995.

[Rodríguez 1998] M. Rodríguez, "Assessment of COTS Microkernel-based System by Fault Injection and Fault Containment with Wrappers", *Proyecto Final de Carrera*, Facultad de Informática de la Universidad Politécnica de Valencia – LAAS-CNRS, Junio 1998.

[Rodríguez *et al.* 2002] F. Rodríguez, J.C. Campelo, J.J. Serrano, "A Memory Overhead Evaluation of the Interleaved Signature Instruction Stream", en *Proceedings 2002 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2002)*, Vancouver (Canadá), pp. 225-232, Noviembre 2002.

[Rosenberg y Shin 1993] H.A. Rosenberg y K.G. Shin, "Software fault injection and its application in distributed systems", en *Proceedings 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pp. 208-217, Toulouse (Francia), Junio 1993.

[Saito *et al.* 1993] T. Saito, H. Aoki, T. Tamaru, N. Owada, "Reliability Improvement in Blanket Tungsten CVD Contact Filling Process for High Aspect Ratio Contact", en *Proceedings 31st International Reliability Physics Symposium (IRPS '93)*, pp. 334-339, Atlanta (Georgia, EE.UU.), Marzo 1993.

[Samson *et al.* 1997] J.R. Samson Jr., W. Moreno, F.J. Falquez, "Validating Fault Tolerant Designs Using Laser Fault Injection (LFI)", en *Proceedings 1997 International Wokshop on Defect and Fault Tolerance in VLSI Systems (DFT'97)*, pp. 175-183, París (Francia), Octubre 1997.

[Samson *et al.* 1998] J.R. Samson Jr., W. Moreno, F.J. Falquez, "A Technique for Automated Validation of Fault Tolerant Designs Using Laser Fault Injection (LFI)", en *Proceedings 28th International Symposium on Fault-Tolerant Computing (FTCS-28)*, pp. 162-167, Munich (Alemania), Junio 1998.

[Santos y Costa 2001] N.D. Santos y D. Costa, "eXception: An Evaluation Tool towards the Demanding Availability of Networking Products", en *FastAbstracts 2001 International Conference on Dependable Systems and Networks (DSN 2001)*, Göteborg (Suecia), Julio 2001.

[Schmid *et al.* 1982] M.E. Schmid, R.L. Trapp, A.E. Davidoff, G.M. Masson, "Upset exposure by means of abstraction verification", en *Proceedings 12th International Symposium on Fault-Tolerant Computing (FTCS-12)*, pp. 237-244, Santa Monica (EE.UU.), Junio 1982.

[Schuette *et al.* 1986] M. Schuette, J. Shen, D. Siewiorek, Y. Zhu, "Experimental evaluation of two concurrent error detection schemes", en *Proceedings 16th International Symposium on Fault-Tolerant Computing (FTCS-16)*, pp. 128-143, Viena (Austria), Julio 1986.

[Schwartz y Melliar-Smith 1983] R.L. Schwartz y P.M. Melliar-Smith, "Specifying and Verifying Ultra-Reliability and Fault-Tolerance Properties", en *Proceedings 26th Computer Society International Conference (COMPCON'83)*, pp. 71-76, San Francisco (California, EE.UU.), Marzo 1983.

[Segall *et al.* 1988] Z. Segall, D. Vrsalovic, D. Soewoprek, D. Yaskin, J. Kownavki, J. Barton, D. Rancey, A. Robinson, T. Lin, "FIAT –Fault Injection based Automated Testing Environment", en *Proceedings 18th International Symposium on Fault-Tolerant Computing (FTCS-18)*, pp. 102-107, Tokio (Japón), Junio 1988.

[Shaw *et al.* 2001] D. Shaw, D. Al-Khalili, C. Rozon, "Accurate CMOS Bridge Fault Modeling With Neural Network-Based VHDL Saboteurs", en *Proceedings 2001 International Conference on Computer-Aided Design (ICCAD'01)*, pp. 531-536, San Jose (California, EE.UU.), Noviembre 2001.

[Shen *et al.* 1985] J.P. Shen, W. Maly, F.J. Ferguson, "Inductive Fault Analysis of MOS Integrated Circuits", *IEEE Design and Test of Computers*, 2(4):13-26, Diciembre 1985.

[Shin *et al.* 1993] H. Shin, N. Jha, XY. Qian, GW Hills, C. Hu, "Plasma Etching Charge-Up Damage to Thin Oxides", *Solid State Technology*, pp. 29-36, Agosto 1993.

[Shirley y Blish 1987] C.G. Shirley y R. Blish, "Thin Film Cracking and Wire Ball Shear in Plastic DIPs Due to Temperature Cycle and Thermal Shock", en *Proceedings 25th International Reliability Physics Symposium (IRPS '87)*, pp. 238-249, San Diego (California, EE.UU.), Abril 1987.

[Shivakumar *et al.* 2002] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, L. Alvisi, "Modeling the Effect of Technology Trends on Soft Error Rate of Combinational Logic", en *Proceedings 2002 International Conference on Dependable Systems and Networks (DSN-2002)*, pp. 389-402, Washington (D.C., EE.UU.), Junio 2002.

[Shoen 1992] J.M. Shoen, "Performance and Fault Modeling with VHDL", ISBN 0-13-658816-6, Prentice-Hall, 1992.

[Sieh 1993] V. Sieh, "Fault Injector using UNIX ptrace Interface", Internal Report nº 11/93, Noviembre 1993.

[Sieh *et al.* 1996] V. Sieh, O. Tsäche, F. Balbach, "VHDL-based Fault Injection with VERIFY", Internal Report nº 5/96, 1996.

[Sieh *et al.* 1997] V. Sieh, O. Tschäche, F. Balbach, "Comparing Different Fault Models Using VERIFY", en *Proceedings 6th International Workshop on Dependable Computing for Critical Applications (DCCA-6)*, pp. 59-76, Grainau (Alemania), Marzo 1997.

[Sieh y Buchacker 2002] V. Sieh y K. Buchacker, "Testing the Fault-Tolerance of Networked Systems", en *Proceedings 2002 International Conference on Architecture of Computing Systems (ARCS 2002)*, pp. 95-105, Karlsruhe (Alemania), Abril 2002.

[Siewiorek 1994] D.P. Siewiorek, "Reliable Computer Systems. Design and Evaluation, 3rd edition", ISBN 15688-1092-X, Digital Press, 1998.

[Siewiorek y Swarz 1982] D.P. Siewiorek y R.S. Swarz, "The Theory and Practice of Reliable System Design", Digital Press, Bedford (Massachusetts, EE.UU.), ISBN 0-93-237613-4, 1982.

[Siewiorek y Swarz 1992] D.P. Siewiorek y R.S. Swarz, "Reliable Computer Systems. Design and Evaluation", 2ª ed., ISBN 1-555-58075-0, Digital Press, 1992.

[Singh y Koren 2003] M. Singh e I. Koren, "Fault Sensitivity Analysis and Reliability Enhancement of Analog-to-Digital Converters", *IEEE Transactions on VLSI Systems*, pendiente de publicación, 2003, disponible en http://euler.ecs.umass.edu/research/siko03.pdf.

[Smith *et al.* 1996] D.T. Smith, B.W. Johnson, J.A. Profeta III, "System Dependability Evaluation via a Fault List Generation", *IEEE Transactions on Computers*, 45(8):974-979, Agosto 1996.

[Srinivasan *et al.* 1994] G.R. Srinivasan, P.C. Murley, H.K. Tang, "Accurate, Predictive Modeling of Soft Error Rate Due to Cosmic Rays and Chip Alpha Radiation", en *Proceedings 32nd International Reliability Physics Symposium (IRPS '94)*, pp. 12-16, San Jose (California, EE.UU.), Abril 1994.

[Stathis 2001] J.H. Stathis, "Physical and Predictive Models of Ultra Thin Oxide Reliability in CMOS Devices and Circuits", en *Proceedings 39th International Reliability Physics Symposium (IRPS '01)*, pp. 132-150, Orlando (Florida, EE.UU.), Abril-Mayo 2001.

[Steininger y Temple 1999] A. Steininger y C. Temple, "Economic Online Self-Test in the Time Triggered Architecture", *IEEE Design & Test of Computers*, pp.81-89, Julio-Setiembre 1999.

[Stewart 1997] B.A. Stewart, "Board Level Automated Fault Injection for Fault Coverage and Diagnosis Efficiency", en *Proceedings International Test Conference (ITC'97)*, pp. 649-654, Washington (D.C., EE.UU.), Noviembre 1997.

[Stott *et al.* 1997] D.T. Stott, M.-C. Hsueh, G.L. Ries,. R.K. Iyer, "Dependability Analysis of a Commercial High-Speed Network", en *Proceedings 27th International Symposium on Fault-Tolerant Computing (FTCS-27)*, pp. 248-257, Seattle (Washington, EE.UU.), Junio 1997.

[Stott *et al.* 1998] D.T. Stott, G.L. Ries,. M.-C. Hsueh, R.K. Iyer, "Dependability Analysis of a High-Speed Network Using Software-Implemented Fault Injection and Simulated Fault Injection", *IEEE Transactions on Computers*, 47(1):108-119, Enero 1998.

[Stott *et al.* 2000] D.T. Stott, B. Floering, D. Burke,. Z. Kalbarczyk, R.K. Iyer, "NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors", en *Proceedings 4th International Computer Performance and Dependability Symposium (IPDS-2K)*, pp. 91-100, Chicago (Illinois, EE.UU.), Marzo 2000.

[Strong *et al.* 1993] A.W. Strong, A.K. Stamper, R. Bolam, T. Furukawa, C. Gow, T. Gow, D.W. Martin, S.W. Mittle, J.S. Nakos, S.L. Pennington, "Gate Dielectric Integrity and Reliability", en *Proceedings 31st International Reliability Physics Symposium (IRPS '93)*, pp. 18-21, Atlanta (Georgia, EE.UU.), Marzo 1993.

[Sylvester y Keutzer 1999] D. Sylvester y K. Keutzer, "Rethinking Deep-Submicron Circuit Design". *IEEE Computer*, pp. 25-33, Noviembre 1999.

[Tammaro 2000] M.J. Tammaro, "The Role of Copper in Electromigration: The Effect of Cu-Vacancy Binding Energy", en *Proceedings 38th International Reliability Physics Symposium (IRPS '00)*, pp. 311-320, San Jose (California, EE.UU.), Abril 2000.

[Texas Instruments 1997] Texas Instruments, "1997 Digital Design Seminar", 1997.

[Tezaki *et al.* 1990] A. Tezaki, T. Mineta, H. Egawa, T. Noguchi, "Measurement of Three Dimensional Stress and Modeling of Stress-Induced Migration Failure in Aluminium Interconnects", en *Proceedings 28th International Reliability Physics Symposium (IRPS '90)*, pp. 221-229, New Orleans (Louisiana, EE.UU.), Marzo 1990.

[Tosaka *et al.* 1999] Y. Tosaka, H. Kanata, S. Satoh, T. Itakura, "Simple Method for Estimating Neutron-Induced Soft Error Rates Based on Modified BGR Model", *IEEE Electron Device Letters*, 20(2):89-91, Febrero 1999.

[Trivedi 1982] K.S. Trivedi, "Probability and Statistics with Reliability, Queuing and Computer Science Applications", ISBN 0-1371-11564-4, Prentice-Hall, 1982.

[Tsai e Iyer 1995a] T.K. Tsai y R.K. Iyer, "FTAPE: a fault injection tool to measure fault tolerance", en *Proceedings 10th AIAA Computing in Aerospace*, pp. 339-346, San Antonio (Texas, EE.UU.), Marzo 1995.

[Tsai e Iyer 1995b] T.K. Tsai y R.K. Iyer, "Measuring fault tolerance with the FTAPE fault-injection tool", en *Proceedings 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/C-C1 2001] "Fault Injection for the Time-Triggered Architecture (FIT)", editado por C. Madritsch, en *Supplement of the 2001 International Conference on Dependable Systems and Networks (DSN 2002), Special Track: European Dependability Initiative*, pp. D-25-D-27, Göteborg (Suecia), Julio 2001.

[TTP/C-C1 2002] TTTech Computertechnik AG, "TTP/C C1 Controller. Specification of the TTP/C C1 Controller", disponible en http://www.tttech.com, 2002.

[Turner y Parsons 1982] T. Turner y R.D. Parsons, "A New Failure Mechanism: Al-Si Bond Pad Whisker Growth During Lifetest", *IEEE Transactions on Component, Hybrids and Manufacturing Technology*, 5:431-435, 1982.

[van der Pol *et al.* 1996] J.A. van der Pol, E.R. Ooms, H.T. Brugman, "Short Loop Monitoring of Metal Step Coverage by Simple Electrical Measurements", en *Proceedings 34th International Reliability Physics Symposium (IRPS '96)*, pp. 148-155, Dallas (Texas, EE.UU.), Abril-Mayo 1996.

[Vargas *et al.* 2000] F. Vargas, A. Amory, R. Velazco, "Estimating Circuit Fault Tolerance by Means of Transient-Fault Injection in VHDL", en *Proceedings 6th International On-Line Testing Workshop (IOLTW'00)*, pp. 67-72, Palma de Mallorca, Julio 2000.

[Velazco *et al.* 1990] R. Velazco, C. Bellon, B. Martinet, "Failure Coverage of Functional test methods: a comparative experimental evaluation", en *Proceedings 1990 International Test Conference (ITC'90)*, Washington (D.C., EE.UU.), Septiembre 1990.

[Velazco y Rezgui 2000] R. Velazco y S. Rezgui, "Transient Bitflip Injection in Microprocessor Embedded Applications", en *Proceedings 6th International On-Line Testing Workshop (IOLTW'00)*, pp. 80-84, Palma de Mallorca, Julio 2000.

[Velazco *et al.* 2000] R. Velazco, S. Rezhui, R. Ecoffet, "Predicting Error Rate for Microprocessor-based Digital Architectures through C.E.U. (Code Emulating Upsets) Injection", *IEEE Transactions on Nuclear Science*, 47(6):2405-2411, Diciembre 2000.

[Velazco *et al.* 2001] R. Velazco, R. Leveugle, O. Calvo, "Upset-like Fault Injection in VHDL Descriptions: A Method and Preliminary Results", *TIMA Research Report ISRN TIMA-RR-01/10-6-FR*, TIMA Laboratory, 2001.

[VFIT 2002] "VFIT User Guide. Revision 1.0", Grupo de Sistemas Tolerantes a Fallos, Universidad Politécnica de Valencia, Julio 2002.

[Walker 2000] M.G. Walker, "Modelling the Wiring of Deep Submicron ICs", *IEEE Spectrum*, 27(3):65-71, Marzo 2000.

[Walker y Thomas 1985] R.A. Walker y D.E. Thomas, "A Model of Design Representation and Synthesis", en *Proceedings 22nd ACM/IEEE Design Automation Conference (DAC '85)*, pp. 453-459, Las Vegas (Nevada, EE.UU.), Junio 1985.

[Woodbury y Shin 1990] M.H. Woodbury y K.G. Shin, "Measurement and analysis of workload effects on fault latency in real time systems", *IEEE Transactions on Software Engineering*, 16(2):212-216, Febrero 1990.

[Young *et al.* 1992] L.T. Young, *et al.*, "A Hybrid Monitor Assisted Fault Injection Environment," en *Proceedings 3rd Dependable Computing for Critical Applications Conference (DCCA-3)*, pp. 281-302, Ed. Springer-Verlag, Palermo (Italia), Septiembre 1992.

[Yount y Siewiorek 1996] C. Yount y D.P. Siewiorek, "A methodology for the rapid injection of transient hardware errors", *IEEE Transactions on Computers*, 45(8):881-891, Agosto 1996.

[Yu 2001] Y. Yu, "A Perspective on the State of Research on Fault Injection Techniques", *Research Report*, Mayo 2001.

[Yuste *et al.* 2003] P. Yuste, D. de Andrés, L. Lemus, J.J. Serrano, P.J. Gil, "INERTE: Integrated NExus-based Real-Time fault injection tool for Embedded systems", en *Proceeding 2003 International Conference on Dependable Systems and Networks (DSN 2003)*, pág. 669, San Francisco (California, EE.UU.), Junio 2003.

[Ziegler 1998] J. Ziegler, "Terrestrial Cosmic Ray Intensities", *IBM Journal of Research and Development*, 42(1), Enero 1998.