UNIVERSIDAD POLITÉCNICA DE VALENCIA

DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

CONTRIBUCIÓN A LA VALIDACIÓN EXPERIMENTAL NO INTRUSIVA DE LA CONFIABILIDAD EN LOS SISTEMAS EMPOTRADOS BASADOS EN COMPONENTES COTS

Tesis Doctoral

Presentada por
D. Juan Pardo Albiach

Dirigida por
Dr. D. Pedro Joaquín Gil Vicente
Dr. D. José Carlos Campelo Rivadulla

Valencia, 2007

A mi mujer Ruth
A mis padres Juan y Leo

Agradecimientos

Estos últimos años han sido de los más importantes, intensos y fascinantes de mi trayectoria profesional. Desde que empecé el camino que me ha llevado a esta tesis doctoral se han sucedido muchas penurias aunque también alegrías y algún que otro cambio de trabajo. Realmente ha sido duro, aunque eso sólo lo sabemos mi familia y yo, puesto que el tener que compaginar diferentes trabajos con tesis doctoral me ha supuesto tener que renunciar a muchas Navidades, Fallas, veranos, Pascuas y demás oportunidades de tiempo libre. Actualmente aún me pregunto si ha valido la pena pero supongo que esta pregunta tendrá respuesta en el futuro, realmente espero que sí. Pero no quisiera quedarme con este sabor de boca, ahora es momento para nuevas ilusiones y proyectos de futuro, cerrar una etapa y empezar otra, y seguir caminando.

Quisiera empezar dando las gracias a la totalidad del grupo de Sistemas Tolerantes a Fallos de la Universidad Politécnica de Valencia, que desde el primer día me acogieron como uno más. En especial quiero dar las gracias a D. Pedro Gil, por haberme tratado siempre con mucho respeto, por estar siempre de buen humor y por haberme enseñado muchas cosas en muy poco tiempo, me gustaría en el futuro poder seguir impregnándome todo lo que sabe. También a D. Juan José Serrano, que en su día fue el primero en invitarme a participar de este proyecto, gracias por asignarme a D. Jose Carlos Campelo como director. A Juan Carlos Ruiz, por haberme enseñado y ayudado en muchas cosas, y también a Pedro Yuste, Juan Carlos Baraza, David de Andrés y Luis Saiz con los que he tenido más contacto, gracias por vuestro trato educado y respetuoso. En última instancia, dar las gracias a la persona más importante de esta tesis, a D. José Carlos Campelo, con el que he trabajado estos últimos años muy estrechamente, al que le he contado todas mis penas, la persona con la que he discutido todos y cada uno de los detalles del presente trabajo, el que me enseñado todo desde el principio, la persona que siempre se ha desvivido por proveerme de todo aquello que he necesitado, el verdadero artífice de la presente tesis doctoral. De verdad muchas gracias de todo corazón, tengo que reconocer que la mayoría de las discusiones que hemos mantenido tomando café, entorno a detalles del presente trabajo, en el 99,9% de las veces tenías razón.

También quiero dar las gracias a D. José Luis Ferrer director de la Escuela Superior de Enseñanzas Técnicas y D. Antonio Falcó director del Departamento de Ciencias Físicas, Matemáticas y de la Computación ambos de la Universidad CEU Cardenal Herrera de Valencia. Gracias Antonio por darme aliento, relajar la presión y permitirme dedicar tiempo y esfuerzo a la tesis. Y por supuesto dar las gracias a Gustavo Salvador vicedecano de la Escuela de Informática y a Nuria Lull mi compañera de despacho, por haberme ayudado, animado, entendido y apoyado en momentos difíciles, espero que seamos amigos siempre.

Finalmente, en el lugar más especial quiero dar las gracias a mi esposa Ruth, por todo lo que me has dado y me sigues dando día a día. Por creer en mi y aguantar muchos fines de semana en casa, por ayudarme constantemente y por todo el cariño que me das, muchas gracias. Y también como no a mis padres, que han sufrido durante muchos años los nervios y la presión de ciertos momentos y que siempre estando ahí, me han demostrado lo mucho que me quieren. Y por supuesto a Jose Antonio, Maria, Mireia y Sofia por ser tan especiales para mi.

A todas las personas que aquí he mencionado muchas gracias de todo corazón.

Juan Pardo
Valencia, 2007

Resumen

Cada día más los computadores están siendo utilizados para una mayor y amplia variedad de aplicaciones y sistemas, y su correcto funcionamiento es a menudo crítico para el éxito del negocio y/o de la seguridad de las personas. Por este motivo el desarrollo o la selección de componentes software de alta calidad, que puedan ser integrados en un sistema, es una cuestión de gran importancia. La utilización de dichos componentes permite agilizar los procesos de desarrollo y reducir por consiguiente los costes asociados. Concretamente estos componentes comúnmente se conocen con el nombre de componentes COTS.

Por otro lado, si se habla de sistemas empotrados de tiempo real que deben ser tolerantes a fallos, la confiabilidad en la integración de dichos componentes debe ser medida. Es por ello la necesidad de encontrar metodologías y técnicas que permitan una evaluación de los mismos en este sentido. Pero tal evaluación debe ser realizada con los sistemas funcionando en un entorno real, para así poder obtener datos que sean representativos del funcionamiento de los mismos y poder certificarlos, como así proponen los estándares y normas internacionales.

Para llevar a cabo dicha evaluación se puede hacer uso de las técnicas de inyección de fallos. Estas técnicas se suelen agrupar en tres tipos, las basadas en inyección de fallos sobre un modelo del sistema, para ello se puede hacer uso de la simulación (SBFI del inglés "*Simulation-Based Fault Injection*"), o mediante la emulación, a partir de un prototipo hardware del sistema. Por otro lado, las que utilizan un hardware específico (HWIFI del inglés "*HardWare Implemented Fault Injection*") y las basadas en software (SWIFI del inglés "*SoftWare Implemented Fault Injection*"). En la presente tesis se ha optado por la última solución, que consiste en emular tanto los fallos hardware y sus consecuencias como los fallos de diseño de los programas mediante software. Las ventajas de estas técnicas son su reducido coste de implementación y su facilidad de automatización. El principal inconveniente es la intrusión que producen en el sistema bajo evaluación. Pero para salvar dicha intrusión es posible recurrir a utilizar los mecanismos de depuración internos que se pueden encontrar en muchos de los microprocesadores actuales. En este sentido la interfaz Nexus™ aparece como idónea para llevar a cabo una inyección de fallos y consiguiente observación del efecto de los mismos sin intrusión alguna en el sistema que se evalúa.

Así en la presente tesis se ha planteado la necesidad de desarrollar una metodología de inyección de fallos que permita verificar el funcionamiento, principalmente ante fallos de diseño del software, de un sistema construido a partir de la unión de diferentes componentes COTS que interactúan entre sí. Y consecuentemente, ha surgido la necesidad de desarrollar una herramienta de inyección de fallos que implemente dicha metodología para evaluar a los sistemas empotrados de tiempo real frente a la aparición de fallos en su funcionamiento real.

Así pues, la herramienta desarrollada ha permitido verificar y validar la metodología propuesta para inyectar fallos de diseño del software haciendo uso de los mecanismos que la interfaz de depuración Nexus™ dispone, para observar en tiempo real el funcionamiento del sistema sin introducir intrusión alguna. Con éstas se han obtenido datos relativos a coberturas de detección de errores, averías del sistema, códigos de error detectados, significado y frecuencia de obtención de los mismos y excepciones lanzadas por el microcontrolador. Finalmente se ha

evaluado información de tipo temporal que ha aportado datos relativos a las averías, desde el punto de vista del dominio del tiempo, viendo los tiempos máximos de ejecución que alcanzaban las tareas cuando eran inyectados fallos en el sistema. De esta manera se han podido calcular los tiempos de detección de errores de los componentes y las posibles averías, por adelanto o retraso, en la entrega del servicio que al final que estos ofrecen.

Abstract

Everyday computers are being used for a greater and ample variety of applications and systems, and its correct operation is often critical for the success of the business and/or the security of the people. For this reason the development or the selection of software components of high quality, which can be integrated in a system, is a question of great importance. The use of these components allows to make agile the development processes and to reduce therefore the associated costs. Concretely these components commonly are known with the name of COTS components.

On the other hand, if we talk about real time embedded systems that must be fault tolerant; the dependability in the integration of these components must be measured. It is for that reason the necessity to find methodologies and techniques that allow an evaluation of such in this sense. But such evaluation must be made with the systems working in real life conditions, thus to be able to collect data that are representative of the operation of such and being able to certify them, as it is proposed in the international standards and norms.

In order to carry out this evaluation fault injection techniques can be used. These techniques usually are grouped in three categories; those based on models of the system like the emulation and simulation (SBFI "*Simulation-Based Fault Injection*"), those that use a specific hardware (HWIFI "*Hardware Implemented Fault Injection*") and those based in the software (SWIFI "*Software Implemented Fault Injection*"). In the present thesis it has been decided on the last solution, which consists of emulating so much hardware faults and its consequences as the software design faults by means of software. The main advantages of these techniques are his reduced cost of implementation and its facility of automatization. The main disadvantage is the intrusion that produces in the system under evaluation. But to overcome this intrusion it is possible to use the internal debugging mechanisms that can be found in many of the present microprocessors. In this sense the Nexus™ emerges as an interesting interface to carry out the fault injection and later observation of the consequences without any intrusion over the system that is evaluated.

Thus in the present thesis the necessity of developing a new methodology to perform a non intrusive fault injection has been considered, all in order to evaluate systems constructed from the union of different COTS components that interact each other. And consequently, the necessity has arisen to develop a fault injection tool that implements this methodology to evaluate real time embedded systems in front of the appearance of faults in its real operation.

Therefore, the developed tool has allowed us to verify and validate the methodology proposed to inject faults doing use of the debugging mechanisms that Nexus™ interface offers to observe in real time the system working. With such tool it has been obtained data about error detention coverage, error codes detected by the system, frequency of exception codes by the microcontroller. Finally temporal data has also been obtained that it has contributed to detect timing failures, seeing the maximum times of execution that reached the tasks when faults were injected in the system. Such a way, it has also been possible to calculate the component error detection latency times.

Resum

Cada dia més els computadors estan sent utilitzats per a una major i àmplia varietat d'aplicacions i sistemes, i el seu correcte funcionament és sovint crític per a l'èxit del negoci i/o de la seguretat de les persones. Per aquest motiu el desenvolupament o la selecció de components software d'alta qualitat, que puguin ser integrats en un sistema, és una qüestió de gran importància. La utilització d'aquestos components permet agilitzar els processos de desenvolupament i reduir per consegüent els costos associats. Concretament estos components comunament es coneixen amb el nom de components COTS. D'altra banda, si es parla de sistemes embarcats de temps real que han de ser tolerants a fallades, la confiabilitat en la integració d'aquestos components ha de ser mesurada. És per això la necessitat de trobar metodologies i tècniques que permeten una avaluació dels mateixos en aquest sentit. Però tal avaluació ha de ser realitzada amb els sistemes funcionant en un entorn real, per a així poder obtenir dades que siguin representatius del funcionament dels mateixos i poder certificar-los, com així proposen els estàndards i normes internacionals.

Per a dur a terme la dita avaluació es pot fer ús de les tècniques d'injecció de fallades. Estes tècniques se solen agrupar en tres tipus: les basades en injecció de fallades sobre un model del sistema, per a això es pot fer ús de la simulació (SBFI de l'anglés *"Simulation-Based Fault Injection"*), o per mitjà de l'emulació, a partir d'un prototip hardware del sistema. D'altra banda, les que utilitzen un hardware específic (HWIFI de l'anglés *"Hardware Implemented Fault Injection"*) i les basades en software (SWIFI de l'anglés *"Software Implemented Fault Injection"*). En la present tesi s'ha optat per l'última solució, que consisteix a emular tant les fallades hardware i les seves conseqüències com les fallades de disseny dels programes per mitjà del software. Els avantatges d'estes tècniques són el seu reduït cost d'implementació i la seva facilitat d'automatització. El principal inconvenient és la intrusió que produeixen en el sistema baix avaluació. Però per a salvar la intrusió és possible recórrer a utilitzar els mecanismes de depuració interns que es poden trobar en molts dels microprocessadors actuals. En aquest sentit la interfície Nexus^{TM} apareix com idònia per a dur a terme una injecció de fallades i consegüent observació de l'efecte dels mateixos sense cap intrusió en el sistema que s'avalua.

Així en la present tesi s'ha plantejat la necessitat de desenvolupar una metodologia d'injecció de fallades que permeta verificar el funcionament, principalment davant de fallades de disseny del software, d'un sistema construït a partir de la unió de diferents components COTS que interactuen entre si. I conseqüentment, ha sorgit la necessitat de desenvolupar una ferramenta d'injecció de fallades que implemente la dita metodologia per a avaluar als sistemes encastats de temps real enfront de l'aparició de fallades en el seu funcionament real. Així, doncs, la ferramenta desenvolupada ha permès verificar i validar la metodologia proposada per a injectar fallades de disseny del software fent ús dels mecanismes que la interfície de depuració Nexus^{TM} disposa, per a observar en temps real el funcionament del sistema sense introduir cap intrusió. Amb estes s'han obtingut dades relatives a cobertures de detecció d'errors, avaries del sistema, codis d'error detectats, significat i freqüència d'obtenció dels mateixos i excepcions llançades pel microcontrolador. Finalment s'ha avaluat informació de tipus temporal que ha aportat dades relatives a les avaries, des del punt de vista del domini del temps, veient els temps màxims d'execució que aconseguien les tasques quan eren injectades fallades en el sistema. D'esta manera s'han pogut calcular els temps de detecció d'errors dels components i les possibles avaries, per avanç o retard, en l'entrega del servei que al final que aquestos ofereixen.

Índice de Contenidos

Índice de Figuras

Índice de Tablas

Capítulo 1

INTRODUCCIÓN

1.1 FUNDAMENTOS Y MOTIVACIÓN

Cada día más los computadores están siendo utilizados para una mayor y amplia variedad de aplicaciones y sistemas, y su correcto funcionamiento es a menudo crítico para el éxito del negocio y/o de la seguridad de las personas. Por este motivo el desarrollo o la selección de componentes software de alta calidad es de gran importancia. La especificación y la evaluación extensiva de la calidad del producto software es un factor clave para asegurar una calidad adecuada. Es importante que cada característica relevante de la calidad del producto software se especifique y se evalúe, utilizando dentro de lo posible, métricas que ya estén validadas o sean ampliamente aceptadas.

La industria del software está entrando en un periodo de madurez, al mismo tiempo que el software se está convirtiendo en un componente esencial de muchos de los productos actuales. Este aspecto de omnipresencia del software lo convierte en un nuevo factor del mercado. Además, con las demandas globales de calidad y seguridad, se está convirtiendo en una necesidad importante el contar con acuerdos internacionales sobre procedimientos de evaluación de la calidad del software.

Se pueden seguir esencialmente dos enfoques para asegurar la calidad de los productos, siendo el primero el aseguramiento del proceso mediante el cual se desarrolla el producto, y el otro la evaluación de la calidad del producto final. Ambos enfoques son importantes y ambos requieren la presencia de un sistema para gestionar y controlar dicha calidad. [ISO/IEC 9126].

El estado del arte en tecnología del software no presenta todavía un esquema descriptivo bien establecido y ampliamente aceptado para evaluar la calidad de los productos software. Numerosas personas han trabajado mucho desde 1976 para definir un marco de referencia para la calidad del software. A lo largo de los años se han adoptado y mejorado los modelos de McCall, de Bohem, de la Armada estadounidense y otros modelos. Sin embargo hoy en día aún es difícil para un usuario o consumidor de productos software entender o comparar la calidad del mismo.

El software nunca se ejecuta aisladamente, sino siempre como parte de un sistema más grande, consistente normalmente en otros productos software con los que tiene interfaces, hardware, operarios humanos y procesos de trabajo. El producto software completo se puede

evaluar por los niveles de las métricas externas escogidas. Estas métricas describen su interacción con el entorno y se evalúan mediante la observación del software en operación. Por otro lado, la calidad en uso del software se puede medir por el grado en que un producto usado por usuarios o sistemas específicos, satisface sus necesidades de alcanzar objetivos concretos con efectividad, productividad, seguridad y satisfacción [ISO/IEC 9126].

En la sociedad actual cada vez cobran más importancia los sistemas informáticos controlando cualquier proceso, desde un sencillo electrodoméstico o un sistema de control de automoción, a los grandes sistemas bancarios o los de telecomunicaciones. En algunas de estas aplicaciones, de su correcto funcionamiento dependen vidas humanas, como en el caso de los sistemas de control de tráfico, equipos de soporte vital o centrales de energía nuclear. Al ser además, muchas de estas aplicaciones de tiempo real, no sólo se debe garantizar que los resultados que se procesan tienen un valor correcto, sino que estos deben ser entregados dentro de los plazos de tiempo especificados. Un entrega tardía de los mismos podría suponer al sistema el producir respuestas obsoletas en el tiempo y en consecuencia actuar de forma incorrecta, y en el peor de los casos provocar serios accidentes.

Desde hace tiempo, la reutilización de software ha venido siendo una práctica común para la construcción de productos software. La reducción de los costes, tiempos y esfuerzos en los procesos de elaboración han sido algunos de los motivos que han llevado a los ingenieros de software a considerar técnicas para la reutilización de partes software en prácticamente cualquier fase del ciclo de vida del producto (análisis, diseño e implementación).

Estas partes software, generalmente, se corresponden con fragmentos de código, procedimientos, librerías y programas desarrollados en otros proyectos o por otros fabricantes, y que pueden ser utilizados para ser incorporados en ciertas partes del nuevo producto que hay que desarrollar. Además, en estos últimos años se ha podido comprobar un aumento en el uso de componentes comerciales en prácticas de reutilización de software. Concretamente, estos componentes comerciales, que comúnmente se conocen con el nombre de componentes COTS (del inglés "*Commercial Off-The-Shelf*"), están siendo considerados con mayor asiduidad para la construcción de sistemas complejos, distribuidos y abiertos. Suelen ser componentes ampliamente extendidos, cuyo precio es significativamente inferior al precio de componentes especialmente diseñados con características similares [Iribarne03].

Por otro lado, el uso de componentes software COTS tiene serios problemas de certificación, debido a que el proceso de diseño y desarrollo de éstos es habitualmente desconocido. Por otro lado, el software COTS está compuesto por componentes de propósito general y suelen poseer pobres especificaciones con respecto a la confiabilidad de los mismos. Puesto que los componentes COTS no se diseñan especialmente para el sistema en el que serán integrados, existe un riesgo residual respecto a aquellas partes del componente no utilizadas por el sistema. También se tiene que tener en cuenta que los componentes COTS tienen un tiempo de vida muy corto, ya que sufren de la aparición de continuas actualizaciones y nuevas versiones, y por tanto no pueden beneficiarse de las estadísticas que definirían su tolerancia a fallos.

Tradicionalmente al realizar el desarrollo de un sistema, por la complejidad intrínseca del propio desarrollo, además del problema de integrar diferentes componentes, la práctica habitual era la de obtener la solución completa que un determinado distribuidor ofrecía en cuanto a hardware y software. Como se ha dicho anteriormente esta situación se va superando y cada día es más habitual recurrir a integrar componentes de diversos fabricantes para la construcción/ensamblaje de los sistemas. Pero la gran pregunta que nos surge ahora es qué componente, de los posibles que me ofrece el mercado para ser incorporado en mi sistema, cumple mejor con la especificación de requisitos que tengo para el propio desarrollo. Y más en

concreto y como ejemplo, si disponemos de una aplicación que se quiere que funcione sobre un sistema operativo de tiempo real (en adelante RTOS, del inglés *"Real-Time Operating System"*), de los diversos que se pueden encontrar en el mercado, ¿cual me dará mejores resultados en cuanto a confiabilidad? Todo esto hace, que sistemas construidos a partir de componentes de muy diverso origen deban ser evaluados frente a la aparición de errores que pudieran provocar averías catastróficas al propio sistema o al entorno en el que trabajan. Es por ello que el sistema debe ser capaz de seguir funcionando, aún a pesar de la aparición de errores, o en su defecto poder auto-reconfigurarse para evitar este tipo de situaciones.

Pero no sólo nos centramos en la parte del software sino que una evaluación integral del sistema nos llevaría a considerar al sistema completo, hardware más software, frente a la aparición de errores en la totalidad de los componentes. Para que el sistema sea capaz de seguir cumpliendo correctamente con su función aún en presencia de fallos y/o errores, se debe haber incluido en el mismo una serie de mecanismos de tolerancia a fallos en la etapa de diseño del sistema. A este tipo de sistemas se les denomina Sistemas Tolerantes a Fallos y permiten al usuario de los mismos depositar una confianza justificada en el servicio que proporcionan. A esta propiedad de los sistemas informáticos se le denomina **confiabilidad** [Laprie92].

Los atributos de la confiabilidad [Laprie92] son las propiedades que nos permiten medirla. Cada uno de ellos se centra en un aspecto particular. Por ejemplo, la disponibilidad se centra en la preparación para el servicio, la fiabilidad en la continuidad del servicio entregado y la inocuidad en la ausencia de consecuencias catastróficas en el entorno. Un ejemplo de sistema de alta disponibilidad es un sistema bancario, en el que el servicio se debe proporcionar el mayor tiempo posible. Un ejemplo de un sistema de alta fiabilidad es un sistema de control de un satélite, ya que una vez está en servicio, éste debe ser proporcionado sin interrupciones; y como ejemplo de sistema de alta seguridad-inocuidad se tiene el sistema de control de una central nuclear, ya que es importante que no se averíe pero aún más que las averías no produzcan consecuencias de tipo catastrófico.

La validación de los atributos de la confiabilidad en un sistema se puede realizar a través de una validación teórica o experimental o combinación de ambas. La validación teórica se lleva a cabo evaluando un modelo teórico del sistema, que generalmente está basado en cadenas de Markov o en algún método de descripción a partir del cual se puedan generar éstas, como cadenas de Petri, redes de actividad estocástica, etc. Algunos ejemplos de validación teórica son los realizados en los trabajos [Campelo99c] [Campelo99d] [Yuste99] [Yuste00] [De Andrés03]. Este tipo de análisis tiene como principal inconveniente que para poder obtener un resultado exacto es necesario utilizar, como entrada al modelo, algunos de los parámetros del sistema difíciles de conseguir de forma teórica. Son fundamentales los siguientes parámetros:

- Coeficientes de cobertura en la detección o recuperación de los errores, que miden la bondad de los mecanismos introducidos de tolerancia a fallos.

- Tiempos de latencia en la propagación de los errores y en la detección o recuperación, que miden la rapidez de los mecanismos de detección de errores y de tolerancia a fallos introducidos.

Para obtener estos parámetros de la manera más exacta posible en un sistema concreto se utiliza la validación experimental.

La validación experimental se puede realizar mediante la observación del sistema en funcionamiento normal, recopilando toda la información que sea posible sobre las averías que sucedan. Este método implica supervisar el comportamiento del sistema hasta que sucedan fallos reales, lo que puede suponer un periodo de tiempo de experimentación muy grande y que

no es factible en la mayoría de los casos. Otro método de validación experimental es la introducción deliberada de fallos en el sistema, también llamada inyección de fallos.

Se han propuesto muchas técnicas para la inyección de fallos. Estas técnicas se suelen agrupar básicamente en tres tipos: las basadas en simulación, las basadas en emulación, las que utilizan un hardware específico y las basadas en software. Las técnicas basadas en simulación utilizan un modelo del sistema e inyectan fallos sobre su simulación; las de emulación inyectan fallos también sobre un modelo que en este caso es un prototipo hardware del sistema. Las técnicas basadas en hardware inyectan fallos físicos a través del hardware. Y la última solución consiste en emular las consecuencias de los fallos hardware y software (los errores) mediante técnicas software. A esta solución se le denomina inyección de fallos implementada por software (o SWIFI, del inglés "*SoftWare Implemented Fault Injection"*).

Por otro lado, dentro del campo de los sistemas empotrados estamos asistiendo a un vertiginoso avance en la tecnología de semiconductores, donde cada vez más aumenta la densidad de integración de los mismos. Se construyen microcontroladores que integran un gran número de periféricos y memorias de diferentes tecnologías y gran capacidad, de forma que el funcionamiento del sistema es cada vez menos visible desde el exterior. Esto hace más difícil diseñar y fabricar herramientas de depuración capaces de implementar las características de control y observación necesarias. Una solución que se ha ido adoptando por diferentes fabricantes ha sido la de incluir dentro del procesador los mecanismos necesarios para conseguir estas capacidades, como es el caso de los sistemas en chip (SoC's del inglés "*System-on-Chip*"). Para estos sistemas existen diferentes interfaces de depuración, que pueden ser utilizados para observar el funcionamiento interno del sistema en tiempo real. Tal es el caso de la interfaz Nexus™ que surge en un intento de unificar criterios por parte de los fabricantes de chips para conseguir una uniformidad en las herramientas de desarrollo en sistemas empotrados.

Teniendo en cuenta que muchos de estos sistemas se desarrollan para aplicaciones de tiempo real, las limitaciones temporales del sistema (por ejemplo, los plazos de ejecución de las tareas) se pueden incumplir por culpa de la intrusión adicional que crea la introducción de la instrumentación de una herramienta de inyección de fallos software. Por ejemplo, en [Campelo01] se describe una herramienta de inyección de fallos software desarrollada dentro del Grupo de Sistemas Tolerantes a Fallos (GSTF). En ese trabajo quedó patente la limitación que tienen las técnicas SWIFI actuales al aplicarlas a los sistemas de tiempo real, que viene dada por un problema conocido como intrusión temporal.

Dicha intrusión temporal se constituye como un importante inconveniente en la evaluación de sistemas, más aún en sistemas de tiempo real con además fuertes restricciones temporales. Por tanto, éste es un problema importante que se debe tener en cuenta siempre que se utilicen herramientas SWIFI en sistemas de tiempo real, de manera que se asegure que no se introduzca ninguna distorsión temporal en la fase de validación u observación. Cualquier perturbación o sobrecarga temporal en la fase de inyección y monitorización de un sistema de tiempo real con fuertes restricciones temporales es inaceptable [Samili04]. De hecho, en [Kopetz97] se define un sistema de tiempo real como un sistema que no sólo debe proporcionar valores correctos en los resultados computados sino que éstos además, deben ser proporcionados en el instante del tiempo requerido. Así la principal característica de un sistema en tiempo real es el cumplimiento de una serie de restricciones temporales denominadas como *"deadlines"*. La pérdida de alguno de estos *deadlines* o plazos temporales podría provocar consecuencias catastróficas.

Por tanto, el estudio frente a fallos de un sistema de estas características, se debe llevar a cabo manteniendo las restricciones temporales como luego en la realidad el sistema debe funcionar. Lo idóneo en estos casos, siempre y cuando sea posible, es utilizar el sistema real frente a modelos o simulaciones del mismo que puedan desestimar los resultados obtenidos. Ya que lo habitual en la mayoría de los casos en los que se utilizan modelos simulados es que se llegue a obtener un modelo normalmente reducido que evita que el sistema sea evaluado en su

completitud frente a la aparición de errores. Por todo ello de ahí la necesidad de encontrar una metodología de inyección de fallos que utilice una interfaz que permita inyectar fallos y observar sus consecuencias de forma totalmente transparente y sin intrusión alguna en sistemas reales empotrados funcionando en tiempo real.

Por último, atendiendo a las normas y estándares internacionales propuestos por organizaciones como ISO o IEC, o incluso a nivel nacional como AENOR, se tiene que normas internacionales como la ISO/IEC 9126-1, que habla del modelo de calidad del producto software, hace mención clara a la necesidad dentro de todo el ciclo de vida de desarrollo de software de una evaluación de la calidad de los productos software desde diferentes perspectivas, viendo atributos externos, donde se mide el comportamiento del código en ejecución y de atributos de la calidad en uso, donde el objetivo es ver que el producto tenga el efecto requerido en un contexto de uso particular. Dentro del modelo de calidad interna y externa que se propone hay un apartado que hace referencia a la confiabilidad del sistema, donde se propone evaluar cuál es el grado de tolerancia a fallos y recuperación del sistema, y en general cuál es su cumplimiento de la confiabilidad. Las métricas externas usan medidas del producto software derivadas de medidas del comportamiento del sistema del que es parte, a través de probar, operar y observar el software ejecutable del sistema.

Por todo ello surge la necesidad de seguir investigando en el campo de la confiabilidad de sistemas, intentando evaluar en condiciones reales de trabajo a los sistemas frente a la aparición de errores, sometiéndolos a diferentes tests y así poder certificarlos en sistemas críticos donde la inocuidad-seguridad de los mismos es un aspecto crucial. Además de evaluar cómo la integración de componentes fabricados por otros y de muy diversa índole puede o no ser beneficioso, añadiendo capas de tolerancia a fallos y en definitiva desarrollar sistemas más seguros evitando que una avería de los mismos pueda suponer pérdidas económicas elevadas o en el peor de los casos riesgo de pérdida de vidas humanas.

1.2 OBJETIVOS DEL PRESENTE TRABAJO

Este trabajo surge a partir de la necesidad de intentar ampliar y contribuir el estado del arte de las técnicas de inyección de fallos actuales, proponiendo una metodología de inyección de fallos software y de observación no intrusivas de los efectos producidos por éstos, que permita evaluar sistemas de tiempo real en condiciones de trabajo reales, donde el software de los mismos pudiera estar desarrollado a partir de la integración de diferentes componentes COTS.

La presente tesis nace como continuación del trabajo llevado a cabo en el desarrollo de la herramienta INERTE [Yuste03e] en el mismo Grupo de Sistemas Tolerantes a Fallos del Departamento de Informática de Sistemas y Computadores de la Universidad Politécnica de Valencia, dentro de los proyectos "*Dbench: Dependability Benchmarking*" (IST-2000-25245) y "Desarrollo de una plataforma de control avanzado de sistemas en motores diésel turboalimentados" (DPI2003-08320-C02-01).

Durante este trabajo se demostró la viabilidad de la utilización de los mecanismos de depuración *on-chip* que ofrecen los microcontroladores actuales para la inyección de fallos. Dentro de estos sistemas se estudió la utilización del estándar Nexus™, debido a que éste ofrece características de observabilidad, portabilidad y sobretodo de muy baja intrusión en la monitorización de sistemas, que hacían posible una inyección de fallos no intrusiva. En este caso se inyectaban fallos de tipo hardware de forma deliberada en zonas de memoria tanto de datos como de código en un intento de validar la propuesta. Ante la necesidad de ir un poco más allá y tener la posibilidad de introducir fallos de tipo software, y más en concreto de fallos de diseño del software o fallos residuales del software, para poder evaluar componentes COTS en

sistemas empotrados, surge la necesidad de profundizar en el estudio de los mecanismos de depuración *on-chip* y más en concreto de la interfaz Nexus™ para obtener una metodología que permita inyectar fallos de todo tipo y observar su efecto tanto desde el punto de vista del dominio del valor como desde el punto de vista del dominio temporal, y así obtener datos objetivos relativos a la robustez de dichos componentes.

Por último y como justificación del trabajo hecho reseñar, que visto el estado del arte de las técnicas y herramientas de inyección de fallos actuales, no hay técnica ni herramienta que no recurra a parar la ejecución [Carreira98] o virtualizar el tiempo del sistema [Rodríguez02] para realizar una inyección de fallos software. Esto no permite realizar un estudio de un sistema de tiempo real en condiciones de trabajo normales, por ello la necesidad de encontrar una técnica y metodología de inyección de fallos que no produzca perturbación alguna, y permita ver la evolución del sistema durante la inyección de fallos de todo tipo (hardware y software) trabajando como lo haría en su funcionamiento normal real.

En resumen, el objetivo principal del presente trabajo es la búsqueda de una metodología de inyección de fallos implementada por software que permita evaluar sistemas de tiempo real construidos a base de integrar componentes software COTS. Dicha metodología debe permitir una evaluación frente a fallos tanto hardware como de diseño del software, que sea transparente desde el punto de vista de la intrusión temporal sobre el sistema a validar. Además se pretende que la técnica sea portable y aplicable a diferentes sistemas que puedan integrar diferentes componentes COTS.

Así los objetivos que se plantean son los siguientes

- Estudiar cómo abordar la problemática de validación de la integración de componentes COTS.

- Estudiar las limitaciones de las técnicas de inyección de fallos actuales al aplicarlas sobre sistemas empotrados de tiempo real.

- Estudiar los mecanismos *on-chip* disponibles actuales que nos permitan conseguir una inyección de fallos sin intrusión temporal. Para ello se profundiza en el estudio del estándar de depuración Nexus™, para ser utilizado como interfaz para la inyección de fallos y observación del sistema.

- Estudiar cómo abordar la inyección de fallos para realizar ensayos de robustez del software. Para ello se evalúa al sistema evitando detener su ejecución durante la experimentación y así respetar las restricciones de tiempo real con el objetivo de poder evaluar sistemas reales bajos condiciones de funcionamiento normales.

- Proponer una metodología de inyección de fallos y observación temporal del sistema, orientada a la evaluación de componentes COTS en sistemas empotrados de tiempo real, aplicable no obstante, a sistemas de propósito general que integren mecanismos OCD (*On-Chip Debugging*).

- Desarrollar una herramienta de inyección de fallos para validar la metodología propuesta, que permita hacer una evaluación integral completa de un sistema constituido por componentes COTS, frente a fallos tanto del hardware como del software. Dicha herramienta debe permitir realizar de forma no intrusiva tanto la inyección de fallos como la posterior observación del funcionamiento del sistema.

1.3 DESARROLLO

A partir de los objetivos mencionados, el desarrollo en capítulos de la presente tesis es el siguiente:

En el capítulo 2 se introducen los términos y aspectos básicos que se utilizan en el campo de la tolerancia a fallos y que sirven de referencia para los siguientes capítulos. Se introducen las funciones de medida de la confiabilidad, qué medios son necesarios para alcanzarla y cómo validarla.

En el capítulo 3 se realiza un estudio del estado del arte en el campo de la inyección de fallos. Se estudian las diferentes técnicas y las variantes de cada una de ellas. Se analizan con más detalle las técnicas SWIFI, por ser más cercanas al objetivo del presente trabajo y se hace una comparación de los trabajos que se han hecho específicamente sobre inyección de fallos en microcontroladores y sobre inyección de fallos en sistemas de tiempo real.

En el capítulo 4 se estudian las capacidades de la interfaz de depuración Nexus™ para la inyección de fallos y posterior observación. En este capítulo se propone una técnica de inyección de fallos software basada en las capacidades que dicha interfaz ofrece, que se beneficia de sus características de portabilidad y observabilidad, y que es capaz de inyectar fallos sobre el sistema sin intrusión espacial ni temporal alguna. Así mismo se explica el modelo de fallo introducido y las medidas a obtener tras los experimentos.

En el capítulo 5 se habla de la necesidad de evaluar la robustez de los componentes COTS, estableciendo en primer lugar una serie de conceptos sobre éstos y cómo evaluar dicha robustez. En este capítulo se propone una metodología de inyección de fallos y posterior observación del efecto de los mismos en sistemas empotrados de tiempo real, aunque aplicable a sistemas empotrados de propósito general.

En el capítulo 6 se describe la herramienta de inyección de fallos desarrollada a partir de la metodología propuesta. Así, se describe el entorno que se ha utilizado para probar las capacidades de la herramienta, viendo los diferentes módulos que la componen y las diferentes posibilidades en la utilización de la interfaz Nexus™. También en este capítulo se estudian las diferentes cargas que se han utilizado en la fase de experimentación.

En el capítulo 7 se muestran las campañas de inyección de fallos llevadas a cabo y los resultados más significativos obtenidos de éstas. Se muestran en tres grandes apartados, similares en su estructura, y que se corresponde con las tres cargas diferentes que se han utilizado. Se muestran resultados referentes a coberturas de detección, tiempos de latencias y tiempos de ejecución de las tareas del sistema con el objetivo de estudiar aspectos temporales consecuentes de los fallos introducidos.

Por último, en el capítulo 8 se comentan las conclusiones que se pueden desprender de la presente tesis y se indican las posibles líneas futuras de trabajo que se pueden seguir a partir de la misma.

Al final el lector podrá encontrar un interesante anexo que profundizan un poco más en ciertos aspectos de la tesis, donde se lleva a cabo un breve resumen de las normas **ISO/IEC 9126-1** e **ISO/IEC 61508,** así como una relación de normas de interés para el presente trabajo, además de una breve descripción de las organizaciones ISO y AENOR.

Capítulo 2

CONCEPTOS BÁSICOS SOBRE TOLERANCIA A FALLOS

2.1 INTRODUCCIÓN

En este capítulo se va a introducir una serie de conceptos que se utilizan en el campo de la tolerancia a fallos. En la bibliografía se utilizan estos conceptos a veces con significados diferentes y a menudo contradictorios. Estas divergencias en la interpretación surgen de las diferentes traducciones al castellano que se hacen de los términos ingleses o franceses. En este sentido, la terminología y definiciones que se aportan en este trabajo siguen las tendencias mostradas en [Avizienis04] y [Gi106] como referentes en el campo de la tolerancia a fallos.

2.2 DEFINICIONES BÁSICAS

La **confiabilidad** se define en [Laprie92] como "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 su o sus usuarios; un usuario es otro sistema (físico o humano) que interacciona con el primero". Así mismo también se dice, que la confiabilidad de un sistema es la capacidad de evitar averías de servicio que sean más frecuentes y más graves de lo aceptable.

Según las aplicaciones a las que está destinado el sistema, se pone énfasis en diferentes facetas de la confiabilidad; lo que quiere decir que la confiabilidad puede ser vista de acuerdo a diferentes, aunque complementarias, propiedades, lo que permite la definición de sus atributos:

- La disposición para un servicio correcto da lugar a la **disponibilidad**.
- La continuidad de un servicio correcto da lugar a la **fiabilidad**.
- La no ocurrencia de consecuencias catastróficas en el entorno da lugar a la **inocuidad**.
- La no ocurrencia de revelaciones no autorizadas de información, da lugar a la **confidencialidad**.
- La ausencia de alteraciones impropias del sistema da lugar a la **integridad**.

- La capacidad para someterse a reparaciones y modificaciones da lugar a la **mantenibilidad**.
- La asociación de la integridad y disponibilidad respecto a acciones autorizadas, junto con la confidencialidad, da lugar a la **seguridad-confidencialidad**.

En la figura 2.1 se puede ver un resumen de las relaciones entre confiabilidad y seguridad en términos de sus principales atributos. La figura no debe ser interpretada como indicación de que, por ejemplo, los desarrolladores de seguridad no tienen interés en la mantenibilidad; o que no se ha hecho nada de investigación en confiabilidad relativa a la confidencialidad. Más bien, indica dónde cae en cada caso el mayor balance de interés y actividad.

La especificación de la confiabilidad y la seguridad de un sistema deben incluir los requisitos relativos a los atributos, en términos de la gravedad y frecuencia aceptable para las averías de servicio para las clases de fallos especificadas y para un entorno de uso dado. Puede suceder para un sistema dado, que uno o más atributos no sean en absoluto requeridos.

Figura 2.1: Atributos de la confiabilidad y la seguridad

Una avería del sistema ocurre cuando el servicio entregado se desvía del cumplimiento de la función del sistema, siendo esta a la que el sistema está destinado. 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.

El desarrollo de sistemas confiables requiere la utilización combinada de un conjunto de medios que pueden ser clasificados como:

- **Prevención de fallos**: Cómo prevenir la ocurrencia o la introducción de fallos.
- **Tolerancia a fallos**: Cómo evitar la ocurrencia de averías de servicio en presencia de fallos.
- **Eliminación de fallos**: Cómo reducir la presencia (número, gravedad) de los fallos.
- **Predicción de fallos**: Cómo estimar el número actual, la incidencia futura y la probabilidad y las consecuencias de los fallos.

La prevención y la tolerancia a fallos tiene como objetivo el proveer la capacidad al sistema de entregar un servicio en el que se pueda confiar; mientras que la eliminación y la predicción de fallos tiene el objetivo de alcanzar la confianza de esta capacidad, justificando que las especificaciones funcionales, de confiabilidad y de seguridad son las adecuadas y que el sistema probablemente las cumpla.

Las nociones que se han introducido pueden ser agrupadas en tres clases como se ve en la siguiente figura 2.2, que definen el esquema de la taxonomía completa de la computación confiable y segura:

- Los **atributos** de la confiabilidad: disponibilidad, fiabilidad, seguridad-inocuidad, confidencialidad, integridad y mantenibilidad; los cuales permiten, en primer lugar,

expresar las propiedades que se esperan de un sistema, y en segundo, valorar la calidad del servicio entregado, que es resultante de la interacción entre los impedimentos y los medios que se oponen a estos.

- Las **amenazas** de la confiabilidad: fallos, errores y averías; que son circunstancias no deseadas, pero no inesperadas en principio, que causan o tienen como resultado la no confiabilidad. Esta no confiabilidad indicará que en adelante no se puede, o no se podrá tener confianza en el servicio ofrecido por el sistema.

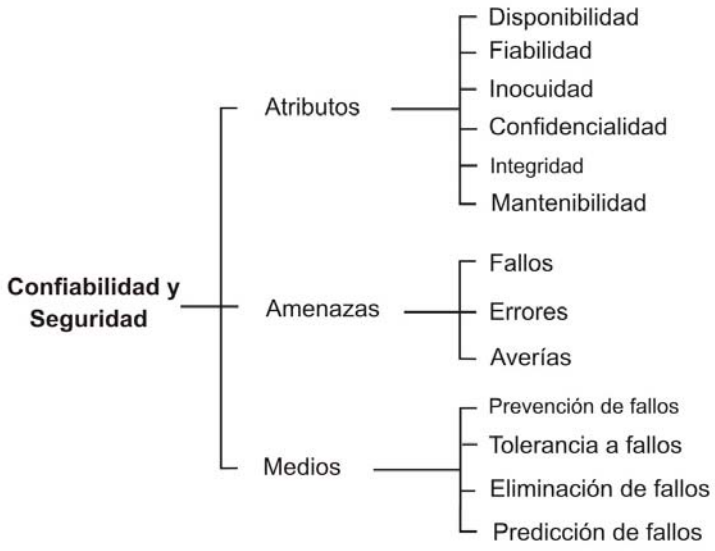


Figura 2.2: Árbol de la Confiabilidad

- Los **medios** para conseguir la confiabilidad: prevención de fallos, tolerancia a fallos, eliminación de fallos, predicción de fallos; que son métodos y técnicas para:

    1. Dar capacidad al sistema para entregar un servicio en el que se pueda confiar.
    2. Tener confianza en esa capacidad.

2.3 ATRIBUTOS DE LA CONFIABILIDAD

Los atributos de la confiabilidad se han definido en la sección anterior de acuerdo con diversas propiedades, sobre las que se puede poner más o menos atención según la aplicación a la que esté destinado el sistema informático considerado:

- La disponibilidad se requiere siempre, aunque a niveles variables, dependientes de la aplicación.
- La fiabilidad, seguridad-inocuidad y confidencialidad, se pueden requerir o no, dependiendo de la aplicación.

La integridad, que se puede definir como la ausencia de alteraciones indebidas de la información, generaliza las definiciones más usuales, relacionadas solamente con la noción de acciones autorizadas (por ejemplo, prevención contra la modificación o borrado no autorizado de la información); naturalmente, cuando un sistema lleva a cabo una política de autorización, “indebido” engloba a “no autorizado”.

La definición dada para **mantenibilidad** va más allá del **mantenimiento correctivo**, dirigido a preservar o a mejorar la capacidad del sistema para entregar un servicio que cumpla con su función (relativo solamente a la reparabilidad); englobado por medio de la evolucionabilidad a las otras formas de mantenimiento: **mantenimiento adaptativo**, que ajusta el sistema a los cambios del entorno, y **mantenimiento aumentativo**, que mejora la función del sistema en respuesta a los cambios definidos por los clientes y los diseñadores, que pueden implicar la eliminación de fallos de especificación. Realmente, la mantenibilidad condiciona la confiabilidad del sistema a lo largo de todo su ciclo de vida, debido a las inevitables evoluciones durante su vida operativa. El término mantenimiento, tal como se ha utilizado aquí sigue siendo el uso común, no solamente incluye las reparaciones, sino también las modificaciones que tengan lugar en el sistema durante la fase de utilización de su ciclo de vida. En la siguiente figura 2.3 puede verse un resumen de las diversas formas de mantenimiento.

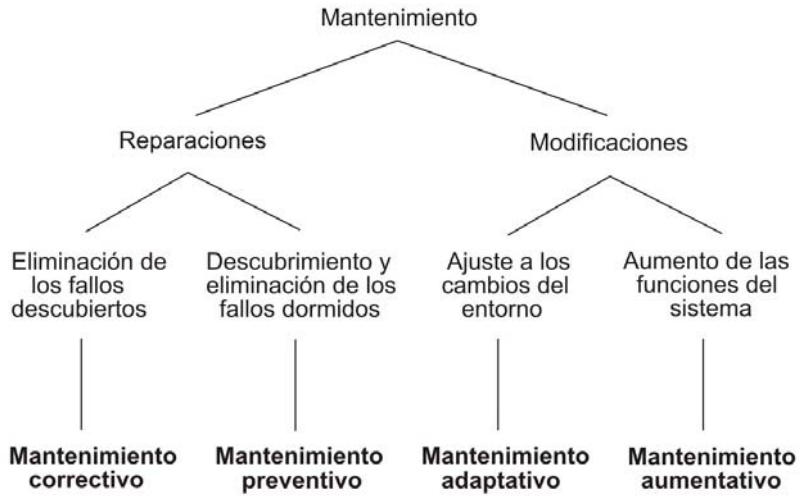


Figura 2.3: Formas distintas de Mantenimiento

La seguridad-confidencialidad no se ha introducido como un atributo de la confiabilidad de acuerdo con la definición usual de la seguridad-confidencialidad que da de ella una noción compuesta, [CEC91] a saber; “la combinación de confidencialidad, prevención contra la revelación no autorizada de información, integridad, prevención contra el enmendado o borrado no autorizado de información, disponibilidad y prevención contra la retención no autorizada de información”.

Según sus definiciones, fiabilidad y disponibilidad enfatizan la capacidad de evitar las averías; mientras que la seguridad-inocuidad enfatiza la capacidad de evitar una clase específica de averías (las averías catastróficas), y la seguridad-confidencialidad la prevención de lo que puede ser visto como una clase específica de fallos (la prevención de acceso y/o manipulación no autorizada de información). Por tanto, la fiabilidad y la disponibilidad están mucho más próximas entre sí que respecto a, por un lado, la seguridad-inocuidad, y por otro, a la seguridad-confidencialidad. Por ello, la fiabilidad y disponibilidad se pueden agrupar conjuntamente, pudiendo definirse globalmente por medio de la minimización de las interrupciones del servicio. Sin embargo este comentario no debería conducir a pensar que la fiabilidad y la disponibilidad no dependen del entorno del sistema: desde hace mucho tiempo se reconoce que la fiabilidad/disponibilidad de un sistema informático está sumamente correlacionada con un perfil de utilización, sea por las averías debidas a fallos físicos, sea por las debidas a fallos de diseño.

El énfasis sobre los diferentes atributos de la confiabilidad incidirá directamente en el tipo de técnicas que se implementen para conseguir que el sistema resultante sea confiable. Sobre todo hay que tener en cuenta que algunos de los atributos son antagónicos, (como por ejemplo

seguridad-inocuidad con disponibilidad) así que se hace necesario encontrar un compromiso. Cuando se consideran las tres dimensiones principales del desarrollo de un sistema informático (costes, prestaciones y confiabilidad) el problema incluso empeora, al no tener tanto dominio de la confiabilidad como de las otras dos.

2.4 IMPEDIMENTOS DE LA CONFIABILIDAD

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

2.4.1 Averías

La ocurrencia de averías se ha definido con respecto a la función del sistema, no respecto a su especificación. Esto es debido a que, en realidad, si se identifica generalmente como avería a un comportamiento inaceptable debido a la no conformidad con la especificación, puede suceder que este comportamiento cumpla con la especificación y sin embargo sea inaceptable para los usuarios del sistema, dejando al descubierto, por tanto, un fallo de especificación. En segundo lugar, el reconocer que el evento es indeseable (y es de hecho una avería), solamente puede ser llevado a cabo después de su ocurrencia, por ejemplo, a través de sus consecuencias.

Un sistema no se avería generalmente siempre de la misma forma. Las formas según las que un sistema puede averiarse son sus modos de avería, que pueden describirse según cuatro puntos de vista: dominio de la avería, detectabilidad de las averías, congruencia de las averías y consecuencias de éstas sobre el entorno. El punto de vista del dominio de la avería, conduce a distinguir entre **averías de valor**, en las que el valor del servicio entregado no cumple con la función del sistema y **averías de tiempo**, en las que el tiempo en el que se entrega el servicio no cumple con la función del sistema.

Estas definiciones generales deben refinarse más. Por ejemplo, la noción de avería de tiempo puede refinarse en **averías con adelanto** y **averías 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 **averías con parada**, que ocurren cuando la actividad del sistema, si es que hay alguna, no es perceptible por sus usuarios.

En relación a cómo el sistema interacciona con sus usuarios, tal ausencia de actividad puede tomar forma de:

a) **Salidas congeladas**: Se entrega un servicio de valor constante, este valor constante puede variar según la aplicación, pudiendo ser, el último valor correcto, algún valor por defecto, etc.
b) **Silencio**: No se entrega servicio alguno en la interfaz de servicio, por ejemplo no se envía ningún mensaje en un sistema distribuido.

Un sistema cuyas averías son solamente averías con parada, es un sistema con parada tras la avería. Las situaciones de salidas congeladas o de silencio dan lugar, respectivamente a los sistemas pasivos tras la avería y a los sistemas con silencio tras la avería.

El punto de vista de la percepción de la avería lleva a distinguir, en caso de varios usuarios, entre **averías consistentes**, en las que todos los usuarios del sistema tienen la misma percepción de las averías y **averías inconsistentes**, en las que los usuarios del sistema pueden percibir de forma diferente alguna avería dada. Las averías inconsistentes son a menudo clasificadas como

averías bizantinas. Hay que resaltar que las averías de un sistema con silencio tras la avería son consistentes, mientras que pueden no serlo en un sistema pasivo tras la avería.

Por otra parte, la gravedad de las averías es el resultado de clasificar las consecuencias de las averías sobre el entorno del sistema, por medio de la ordenación de los modos de avería según diferentes niveles de gravedad, a los que se asocia generalmente las probabilidades máximas de ocurrencia permisibles. El número, el etiquetado 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 menores**, donde las consecuencias son de un orden de magnitud igual que el beneficio obtenido por el servicio entregado en ausencia de averías.
- **Averías catastróficas**, donde las consecuencias son inconmensurablemente superiores al beneficio obtenido por el servicio entregado en ausencia de averías.

Un sistema en donde todas las averías son, en una medida aceptable, menores o benignas, es un **sistema seguro tras la avería**.

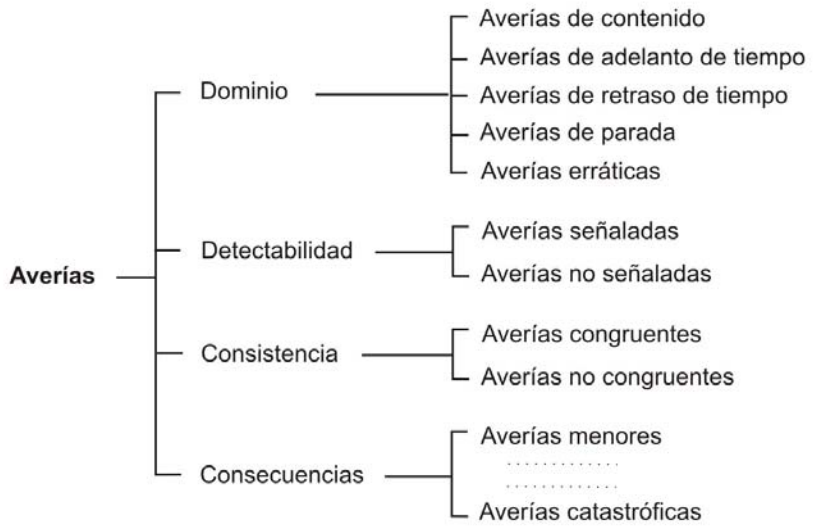


Figura 2.4: Clases de Averías

La noción de gravedad de las averías permite definir la noción de criticidad: la **criticidad** de un sistema es la mayor gravedad de sus posibles modos de avería. La relación entre los modos de avería y la gravedad de las averías es muy dependiente de la aplicación considerada. Sin embargo, existe una amplia clase de aplicaciones donde la inactividad se considera como una posición segura por naturaleza (por ejemplo, transportes terrestres o producción de energía), 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 (tanto los pasivos como con silencio tras avería) y los sistemas seguros tras avería son, sin embargo, ejemplos de sistemas controlados tras avería, es decir, sistemas diseñados e implementados con el fin de que se averíen solamente, o lo hagan 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 posean silencio tras la avería en vez de balbucear o que posean averías consistentes en lugar de inconsistentes. Los sistemas

controlados tras la avería pueden ser definidos, además, imponiendo alguna condición al estado interno o a la accesibilidad, al igual que los sistemas llamados "**fail-stop**".

2.4.2 Errores

El error se define como la parte del estado total de un sistema que puede conducir a una avería, así pues una avería ocurre cuando el error hace que el servicio entregado se desvíe del servicio correcto. A la causa del error se le denomina fallo. Un error es detectado si se señala su presencia por medio de un mensaje de error o una señal de error. Los errores que están presentes pero que no son detectados se denominan *errores latentes*. Puesto que un sistema consiste en una serie de componentes que interaccionan, el estado total del sistema es el conjunto de estados de sus componentes. La definición implica que un fallo causa originalmente un error dentro del estado de uno o más componentes, pero la avería del servicio no ocurrirá mientras el estado externo de ese componente no forme parte del estado externo del sistema. Una vez que el error se convierta en parte del estado externo del componente, ocurrirá una avería de servicio de ese componente, pero el error todavía será interno al sistema completo. El hecho de que un error conduzca o no a una avería depende de tres factores principales:

a) De la composición del sistema, y particularmente de la naturaleza de la redundancia existente:

   - **Redundancia intencionada o extrínseca** es la introducida para tolerar los fallos. Está destinada explícitamente a evitar que un error de lugar a una avería.

   - **Redundancia no intencionada o intrínseca** que puede tener el mismo efecto, aunque inesperado, que la redundancia intencionada. En la práctica es a veces muy difícil, si no imposible, construir un sistema sin alguna forma de redundancia.

b) De la actividad del sistema, ya que una palabra con error puede ser rescrita correctamente antes de que produzca daños.

c) De la definición de avería desde el punto de vista del usuario, ya que lo que es una avería para un usuario dado puede no ser más que una molestia soportable para otro.

Una clasificación conveniente de errores consiste en describirlos en términos de las averías elementales de servicio que causa, así se tiene errores de contenido versus errores de temporización, errores detectados versus errores latentes, errores coherentes versus errores incoherentes cuando el servicio va a dos o más usuarios, errores de menor importancia versus errores catastróficos.

2.4.3 Fallos

Todos los fallos que pueden afectar a un sistema durante su vida se clasificarán de acuerdo ocho puntos de vista básicos, que da lugar a las clases de *fallos elementales*, como se ve la figura 2.5.

Las causas fenomenológicas nos llevan a distinguir entre:

- **Fallos físicos**, que son debidos a fenómenos físicos adversos.
- **Fallos humanos**, que resultan de imperfecciones humanas.

La naturaleza de los fallos conduce a distinguir entre:

- **Fallos no deliberados**, que aparecen o son creados de manera fortuita.
- **Fallos deliberados**, resultantes de decisiones dañinas, con o sin intención maligna.

La fase de creación con respecto a la vida del sistema lleva a distinguir entre:

- **Fallos de desarrollo**, resultantes de imperfecciones originadas, bien en el desarrollo del sistema (de la especificación de las necesidades a la implementación) o durante las modificaciones posteriores, o bien durante el establecimiento de los procedimientos de explotación y mantenimiento del sistema.
- **Fallos de operación**, que aparecen durante la explotación del sistema.

Las fronteras del sistema llevan a distinguir entre:

- **Fallos internos**, que son aquellas partes del estado del sistema que, cuando se invoquen por la actividad computacional producirán un error.
- **Fallos externos**, que son el resultado de interferencias o de la interacción con su entorno físico.

Según la dimensión de los fallos se tiene:

- **Fallos de hardware**: que son aquellos originados en el propio hardware o que afectan a éste.
- **Fallos de software**: que afectan al software, es decir al programa o a los datos.

El objetivo de los fallos lleva a distinguir entre:

- **Fallos maliciosos**: cuando son introducidos por humanos bien durante el desarrollo del sistema o bien directamente durante su uso con el objetivo de causar algún daño al sistema.
- **Fallos no maliciosos**: cuando éstos son introducidos sin el objetivo de dañar.

La persistencia temporal de los fallos conduce a distinguir entre:

- **Fallos permanentes**, cuya presencia no está ligada a condiciones puntuales, sean internas como la actividad computacional o externas, como el entorno.
- **Fallos temporales**, cuya presencia está ligada a aquellas condiciones y están, por tanto, presentes un tiempo limitado.

La noción de fallo temporal merece los siguientes comentarios: los fallos temporales externos que son originados por el entorno físico son denominados a menudo fallos transitorios. Por otra parte los fallos temporales internos son llamados a menudo fallos intermitentes, que son el resultado de la presencia de combinaciones o condiciones que se dan raramente. Ejemplos de estos son los fallos “sensibles a patrones” en las memorias semiconductoras, que son situaciones que suceden cuando la carga del sistema sobrepasa un determinado nivel, como temporizaciones o sincronizaciones críticas.

Los fallos agrupados bajo la denominación de fallos físicos incluyen a todos los fallos que tienen un origen físico como fallos elementales. Aquí se agrupan los fallos debidos a averías operacionales y los fallos de desarrollo con un origen de producción en diferentes copias de un sistema o componente hardware. A estos últimos se les suele llamar fallos de producción. Mientras que los fallos operacionales físicos pueden ser internos o externos, los fallos de producción sólo pueden ser internos.

Los fallos humanos se corresponden con cuatro clases de fallos combinados:

- **Fallos de diseño**, que son fallos de desarrollo, accidentales o intencionados no malignos.
- **Fallos de interacción**, que son fallos externos de operación, accidentales o intencionados no malignos.
- **Lógica maligna o malévola**, que son fallos internos, intencionados malignos.
- **Intrusiones**, que son fallos externos de operación, intencionados malignos.

Si fuesen posibles todas las combinaciones de estas ocho clases de fallos, habría un total de 256 clases de fallos combinados. Sin embargo, todos los criterios no son aplicables a todas las clases de fallos; por ejemplo los fallos naturales no pueden clasificarse según su objetivo, intención o capacidad. Se han identificado 31 combinaciones probables, como puede verse en la figura 2.6. En el futuro se pueden identificar más combinaciones.

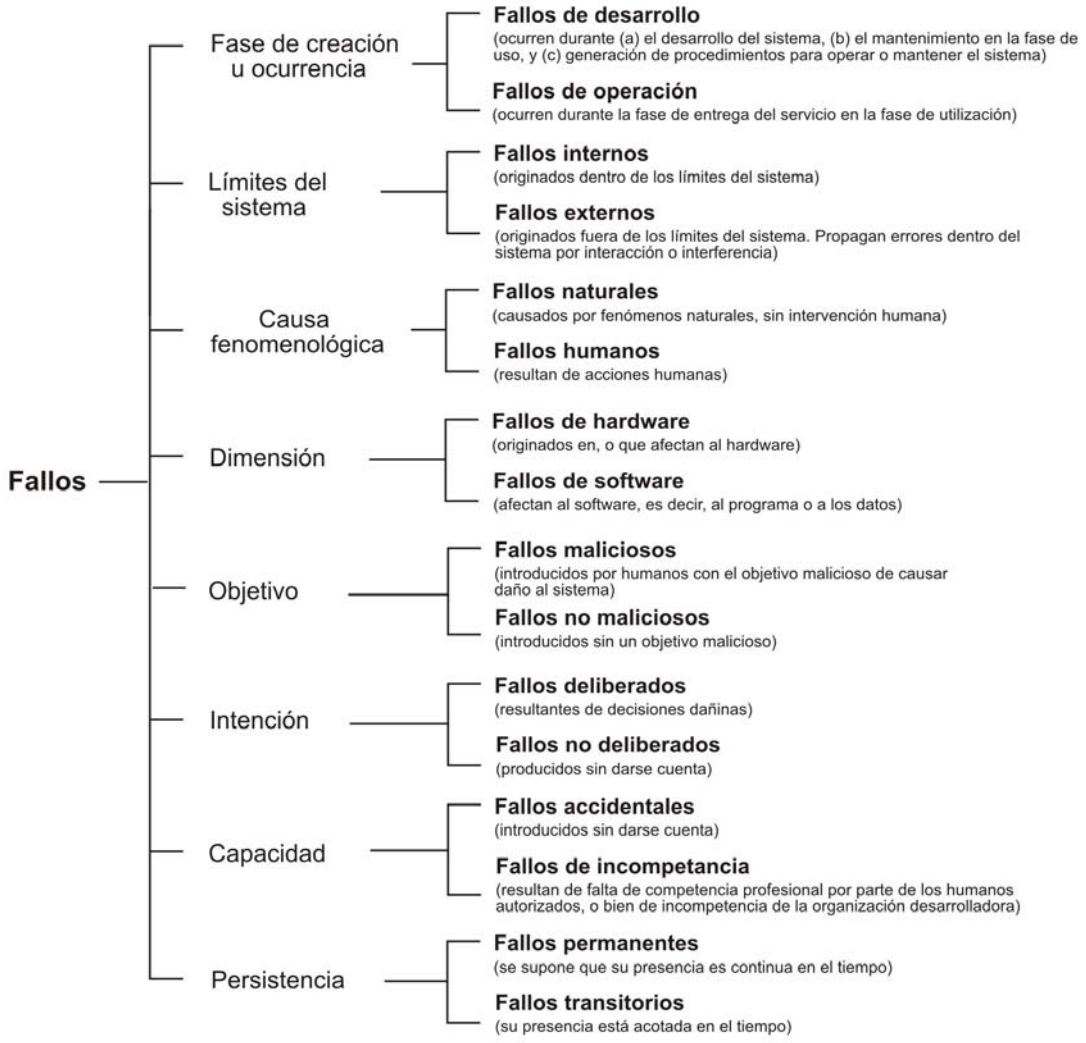
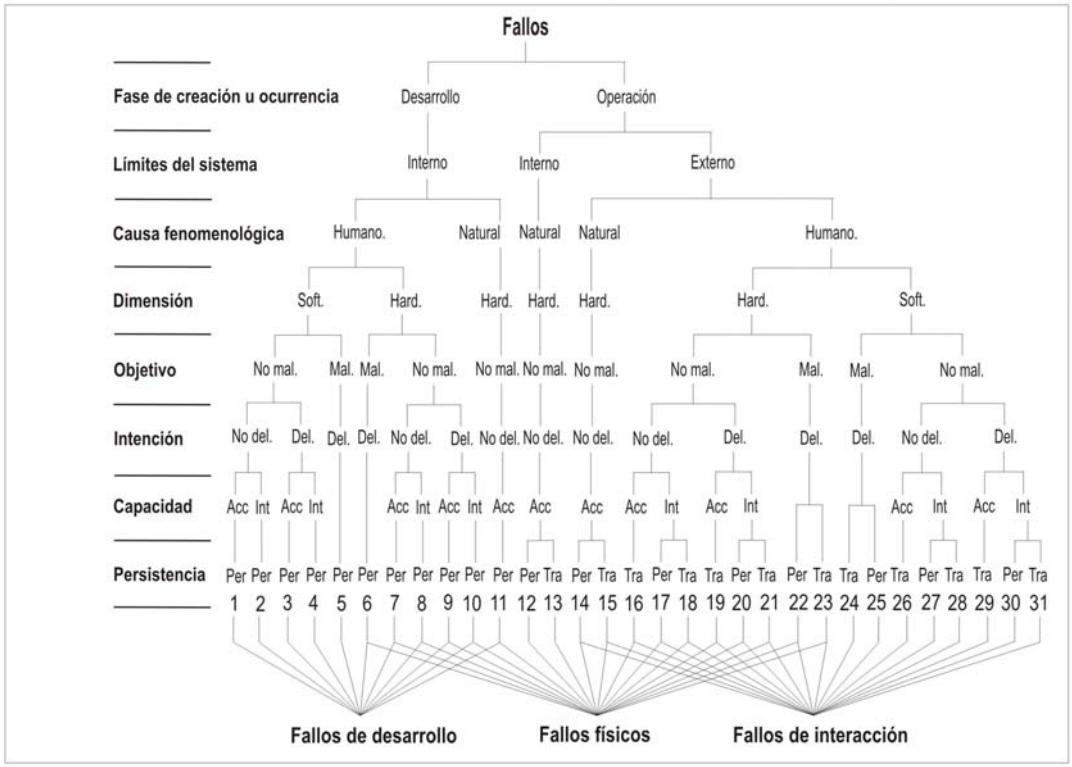


Figura 2.5: Clases de fallos elementales

(a)



(b)

Figura 2.6: Clases de fallos combinados. (a) Representación matricial. (b) Representación en árbol

2.4.4 Patología de los fallos

Los mecanismos de creación y manifestación de los fallos, errores y averías se pueden resumir como sigue:

1. Un fallo es *activo* cuando produce un error. Si no, es un fallo *inactivo*. Un fallo activo puede ser 1) un fallo interno que era previamente inactivo y que ha sido activado por el proceso de cómputo o por condiciones ambientales, o bien 2) un fallo externo. La activación del fallo es la aplicación de una entrada (el patrón de la activación) a un componente que hace que el fallo inactivo se convierta en activo. La mayoría de los fallos internos están conmutando entre sus estados inactivo y activo.

2. La propagación del error dentro del componente dado (es decir, la propagación *interna*) es causada por el proceso de cómputo: el error se transforma sucesivamente en otros errores. La propagación del error de un componente A a otro componente B que recibe servicio de A (es decir, la propagación *externa*) ocurre cuando, a través de la propagación interna, un error alcanza la interfaz de servicio del componente A. En este momento, el servicio entregado por A a B pasa a ser incorrecto, y la avería de servicio de A que sobreviene, aparece como un fallo externo a B, propagándose el error dentro de B vía su interfaz de utilización.

3. Una avería de servicio ocurre cuando el error se propaga a la interfaz de servicio y causa que el servicio entregado por el sistema se desvíe del servicio correcto. La avería de un componente causa un fallo permanente o transitorio en el sistema que contiene este componente. La avería de servicio de un sistema causa un fallo externo permanente o transitorio para el otro sistema que recibe servicio del sistema dado.

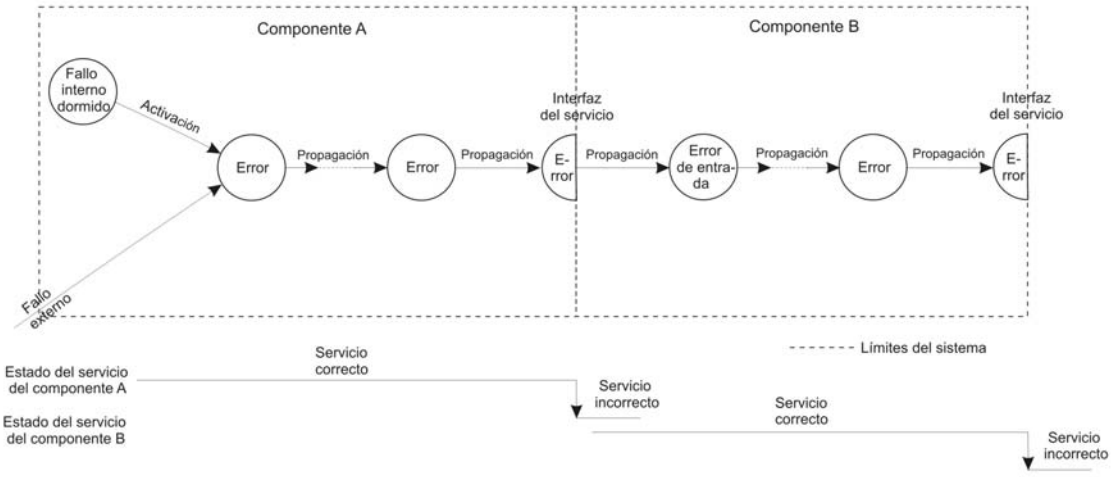


Figura 2.7: Propagación de los errores

Estos mecanismos permiten completar la "***cadena fundamental***" que sigue:

Figura 2.8: Cadena fundamental de amenazas de la confiabilidad y la seguridad

Las flechas de esta cadena expresan la relación de causalidad entre fallos, errores y averías. No deben ser interpretadas de forma restringida, ya que mediante propagación pueden generarse varios errores antes de que se produzca una avería, y un error puede provocar un fallo sin que se haya observado una avería (en caso de que no se realice esta observación), ya que una avería es un evento que ocurre en el interfaz entre dos componentes. La transformación entre los estados de fallo, error y avería no se produce de manera simultánea 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. El tener en cuenta estos casos lleva a distinguir entre fallos independientes, que son atribuidos a diferentes causas 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 estos últimos produzcan a veces errores similares. Los errores similares causan averías de modo común. La generalización de la noción de errores parecidos da lugar a la noción de errores coincidentes, esto quiere decir, errores creados a partir de la misma entrada. La relación temporal entre las averías múltiples lleva a distinguir entre las averías simultáneas, que ocurren en la misma ventana de tiempo predefinida y averías secuenciales, que no ocurren en la misma ventana de tiempo predefinida.

2.5 MEDIOS PARA CONSEGUIR CONFIABILIDAD

En este punto se va a examinar que es la tolerancia a fallos, la eliminación de fallos y la predicción de fallos. La prevención de fallos no se va a tratar, ya que claramente se refiere a la ingeniería general de sistemas. Este punto termina con una discusión sobre la relación entre los medios para la confiabilidad.

2.5.1 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 eliminar los errores del estado computacional, a ser posible antes de que ocurra una avería. El tratamiento de los fallos está destinado a prevenir que se activen uno o varios fallos de nuevo. El procesamiento de los errores se puede realizar por medio de tres principios de diseño:

- Mediante la **detección de errores**, que permite identificar como tal a un estado erróneo.
- Mediante el **diagnóstico de errores**, que permite apreciar los daños producidos por el error detectado, o por los propagados antes de la detección.
- Mediante la **recuperación de errores**, donde se sustituye el estado erróneo por otro libre de errores. Esta sustitución puede hacerse de tres formas:

    a) Mediante la recuperación hacia atrás, en donde el estado erróneo se sustituye por otro ya sucedido antes de la ocurrencia del error. Este método incluye el establecimiento de puntos de recuperación, que son instantes durante la ejecución de un proceso donde se salvaguarda el estado, pudiendo éste posteriormente ser restaurado tras la ocurrencia del error.

b) Mediante la recuperación hacia adelante, donde la transformación del estado erróneo consiste en encontrar un nuevo estado a partir del cual el sistema pueda seguir funcionando (frecuentemente en modo degradado).

c) Mediante la compensación, donde el estado erróneo contiene 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 muy diferente según la técnica de recuperación de errores adoptada. En la recuperación de errores hacia delante 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 por 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 estructural.

El primer paso en el tratamiento de los fallos es el **diagnóstico de los fallos**, 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 mismos, o sea, hacerlos pasivos, es decir, realizar una **pasivación de los fallos**. Este objetivo se lleva a cabo eliminando del proceso de ejecución posterior, los elementos considerados como defectuosos. Si el sistema no es capaz de seguir dando el servicio anterior, puede tener lugar una **reconfiguración**, que consiste en la modificación de la estructura del sistema, para que los componentes no averiados del mismo permitan la entrega de un servicio aceptable, aunque degradado. Una reconfiguración puede implicar la renuncia a algunas tareas, o una reasignación de éstas entre los componentes no averiados.

La pasivación de 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 como un **fallo blando**, o dulce. Si se realiza la pasivación, el fallo se considera un **fallo duro** o sólido. A primera vista, las nociones de fallos blandos y duros parecen sinónimas de las de fallo temporal y permanente respectivamente. 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 si mismo siempre que no se haya generado un fallo permanente por propagación del temporal. De hecho las nociones de fallo blando y duro son útiles por los motivos siguientes:

- Distinguir un fallo temporal de uno permanente es una tarea difícil y 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, y fallos de diferentes clases pueden dar lugar a errores similares. Por lo tanto la noción de fallo blando y duro incorpora 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 estas nociones 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 sobre elementos de memoria es un fallo temporal?. Sin embargo, un fallo de este tipo es claramente un fallo blando.

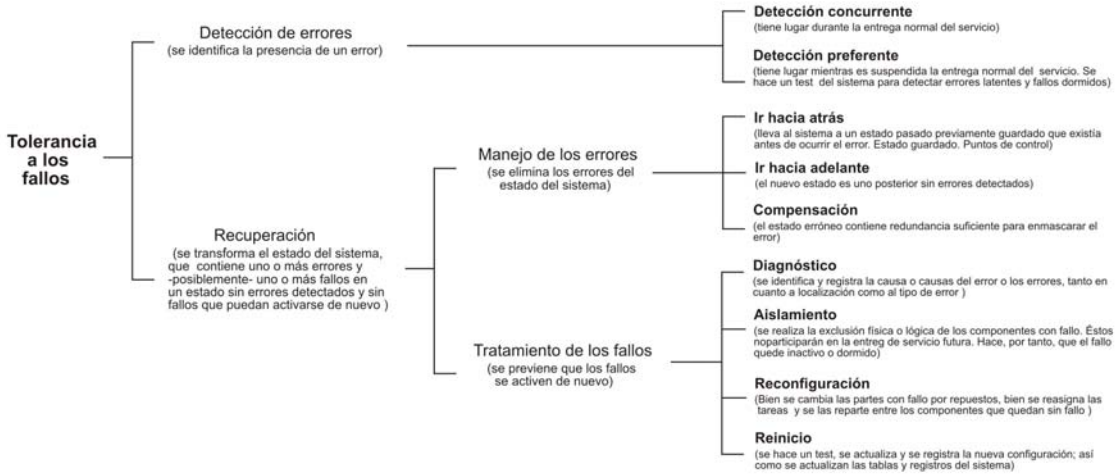


Figura 2.9: Técnicas de tolerancia a fallos

2.5.2 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 se deben realizar 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 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 formas:

- Condiciones generales, que se aplican a una clase de sistema dado, y que son en consecuencia, (relativamente) independientes de las especificaciones, por ejemplo ausencia de bloqueos o conformidad con las reglas de diseño y de realización.
- Condiciones específicas del sistema considerado, deducidas directamente de su especificación.

Las técnicas de verificación se pueden clasificar según si implican o no la activación del sistema. La verificación de un sistema sin su activación real se denomina verificación estática, que se puede realizar de las siguientes formas:

- 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 (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 previa activación se denomina **verificación 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 el caso de los tests de verificación también llamados simplemente tests.

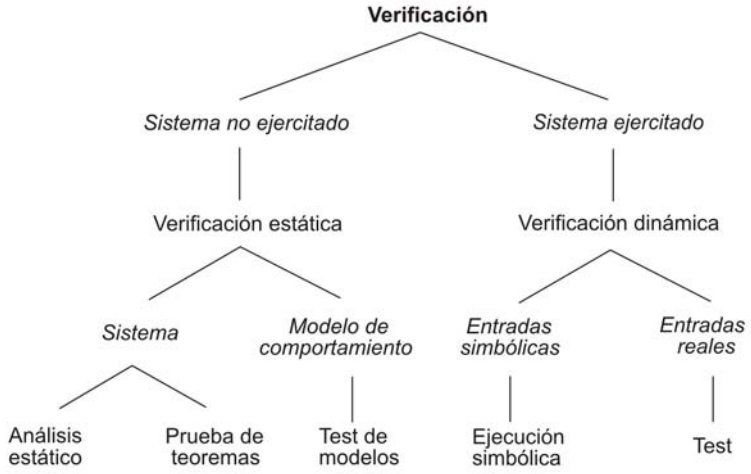


Figura 2.10: Enfoques de verificación

El test exhaustivo de un sistema respecto de todas sus entradas posibles, es generalmente impracticable. Los métodos para determinar los patrones particulares de test pueden clasificarse de acuerdo a dos puntos de vista: según los criterios de selección de las entradas de test y según la generación de las entradas de test. En la figura 2.11 puede verse un resumen de los diferentes enfoques para testear, de acuerdo a los criterios de selección anteriores. En la parte superior de la figura se identifican los enfoques de test elementales. La parte inferior muestra las combinaciones de estos enfoques elementales, en donde se hace la distinción entre test hardware y software, ya que el test de hardware está orientado normalmente a los fallos de producción, y el de software a los fallos de desarrollo.

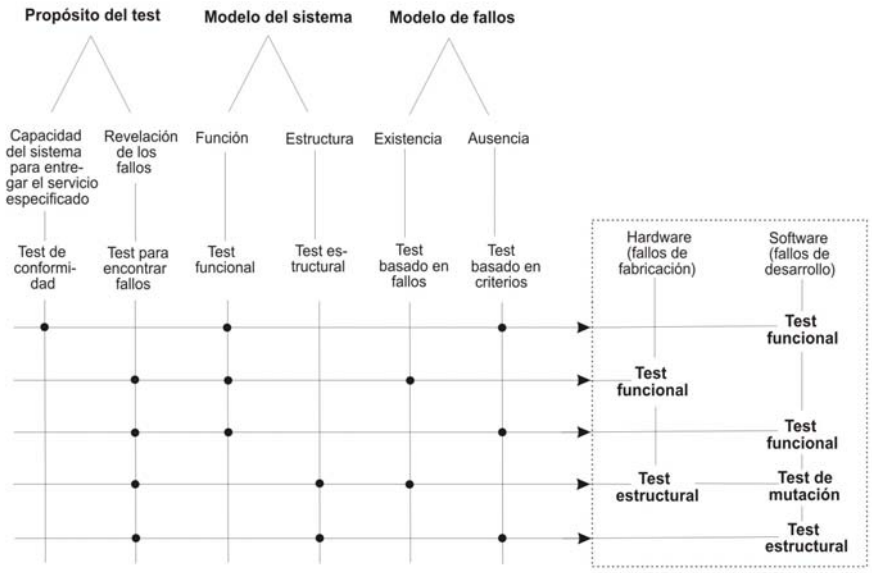


Figura 2.11: Enfoques de test según la selección de los patrones de test

Es de especial importancia también, respecto a lo que el sistema no debe hacer, el verificar que el sistema no hace nada más de lo que está especificado. Este detalle está relacionado con la seguridad-inocuidad y con la seguridad-confidencialidad.

Se denomina **diseño orientado a la verificación**, al diseño de sistemas de forma que sea fácil su verificación. Este planteamiento esté especialmente desarrollado con respecto a los fallos físicos en el hardware, llamándose en este caso particular diseño para el test.

Se llama **mantenimiento correctivo** a la eliminación de fallos durante la vida operativa de un sistema. El mantenimiento correctivo puede ser de dos formas:

- *Mantenimiento curativo*, destinado a eliminar los fallos que han producido uno o más errores que se han detectado.
- *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 se pueden mantener en línea (sin interrupción del servicio), o fuera de línea. Es importante notar, finalmente, que la frontera entre mantenimiento correctivo y tratamiento de los fallos es relativamente arbitraria. En particular, el mantenimiento curativo se puede considerar un medio de tolerancia a fallos.

2.5.3 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. La evaluación tiene dos facetas:

- Evaluación **cualitativa**, destinada, en primer lugar a identificar, clasificar y ordenar los modos de avería. Y en segundo lugar a identificar las combinaciones de eventos (averías de componentes o condiciones del entorno) que dan lugar a sucesos no deseados.

- Evaluación **cuantitativa**, destinada a la evaluación, 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 que permiten la evaluación cualitativa y la cuantitativa son específicos de cada modo de evaluación (por ejemplo, análisis de modos y efectos de las averías para la evaluación cualitativa o cadenas de Markov para la cuantitativa) o sirven para los dos 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 aquel en donde el servicio entregado cumple con la función del sistema. *Servicio incorrecto* será aquel en donde el servicio entregado no cumple con la función del sistema.

Una avería es, por tanto, una transición entre el servicio correcto y el incorrecto. A la transición entre el servicio incorrecto y el correcto se le 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, respecto a la alternancia entre servicio correcto y servicio incorrecto.

Habitualmente también se considera una tercera medida, la **mantenibilidad**, que se puede definir como la medida del tiempo de restauración tras la última avería, o de manera equivalente, de la entrega continua de un servicio incorrecto. La **seguridad-inocuidad**, como medida, puede verse como una extensión de la fiabilidad. Si se agrupan el estado de servicio correcto con el estado de servicio incorrecto posterior a las averías benignas como un estado seguro (en el sentido de la ausencia de daños catastróficos, no de peligro), 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, se pueden distinguir diversos servicios, así como diversas formas de entregar el servicio. Desde la plena capacidad hasta la completa parada. Lo que puede ser visto como entregas de servicio cada vez menos correctas. La medida combinada de prestaciones y confiabilidad se denomina habitualmente **prestabilidad**.

Los dos principales métodos de la predicción de fallos probabilística, destinados a la obtención de estimadores cuantificados de las medidas de confiabilidad son el modelado y el test (de evaluación). Estos métodos son complementarios directamente en el sentido de que el modelado precisa de datos relativos a los procesos elementales modelados (procesos de avería, de mantenimiento, de activación del sistema, etc.) que se pueden obtener mediante test.

Cuando se realiza una evaluación mediante modelado, los métodos a utilizar difieren significativamente según que el sistema se considere con la fiabilidad estable o con crecimiento de ésta. Estas últimas características se pueden definir de la siguiente forma:

- **Fiabilidad estable**: cuando se preserva la capacidad el 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 fase de 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 la fase de procesamiento del modelo para la obtención de las expresiones y valores de las medidas de la confiabilidad del sistema.

La evaluación se puede realizar respecto a los fallos físicos, los fallos de diseño, o una combinación de ambos. La confiabilidad de un sistema es altamente dependiente de su entorno, tanto en el sentido amplio del término, como en el significado más restringido de su carga.

Los modelos de la 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.

Cuando se evalúa un sistema tolerante a fallos, la cobertura de los mecanismos de procesamiento de los errores y de tratamiento de los fallos tiene una influencia primordial. Su evaluación se puede hacer mediante modelado o mediante test. A este tipo de test se le denomina inyección de fallos.

2.5.4 Dependencias entre los medios para alcanzar la Confiabilidad

En las definiciones de Prevención, Tolerancia, Eliminación y Predicción de fallos del apartado 2.2, 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 secundarios en el sistema.

Por este motivo, es necesaria la utilización combinada de varios de estos métodos para lograr un sistema de alta confiabilidad. 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 hacer notar que el proceso es aún más recursivo, 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 alta confiabilidad, y así sucesivamente.

Los anteriores razonamientos ilustran la fuerte interacción 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 de simulación o un prototipo).

2.6 CONFIABILIDAD Y TOLERANCIA A FALLOS

En lo que concierne al desarrollo de sistemas de alta confiabilidad, 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 [Arlat90].

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

de la complejidad tecnológica actual, la Tolerancia a fallos mantiene un papel preponderante en la búsqueda de un nivel significativo de confiabilidad. La introducción de la Tolerancia a fallos en el proceso de desarrollo de un sistema informático hace necesario:

- Determinar los tipos de fallos susceptibles de activarse en la fase operativa.
- Diseñar el sistema de manera que utilice mecanismos de redundancia para reducir los efectos de dichos fallos sobre el servicio proporcionado en la fase operativa.

2.7 CONFIABILIDAD Y VALIDACIÓN

La validación constituye uno de los principales problemas asociados al desarrollo y explotación de sistemas informáticos de alta confiabilidad. 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?" [Laprie85].

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 2.5.

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.

2.8 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 [Arlat90] [Avizienis04] [Gil06]:

- 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 específicos 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 [Bouricius69], en alusión a un nivel de detalle que sólo tiene en cuenta los procesos de ocurrencia de fallos y reparaciones en los componentes del sistema, hasta los que consideran el comportamiento de los sistemas de tratamiento de fallos, que se denominan microscópicos [Dugan89].

Los modelos macroscópicos se pueden resolver utilizando datos estadísticos de los fabricantes de los circuitos del sistema, teniendo el gran inconveniente de la inexactitud de sus resultados, dado el tratamiento demasiado superficial del proceso de ocurrencia de los fallos. Además, su aplicación para el cálculo de la confiabilidad del software del sistema es muy compleja [Kanoun89].

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).

2.9 VALIDACIÓN EXPERIMENTAL E INYECCIÓN DE FALLOS

La validación experimental puede llevarse a cabo de dos formas diferentes [Arlat90]:

- 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. 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 [Arlat90]:

> *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.

En la figura 2.12 [Arlat90] [Dbench04] [Gil92] se puede observar el proceso de validación de un sistema tolerante a fallos, tanto desde el punto de vista teórico como experimental, mediante inyección de fallos. Como se aprecia en la figura, para efectuar una validación teórica se seguirá el camino (1) del organigrama (I). A partir de unas especificaciones del sistema a desarrollar se construye, en primer lugar, un modelo teórico, normalmente basado en cadenas de Markov o Redes de Petri. A continuación se tiene que caracterizar el modelo añadiéndole las

coberturas y tiempos de latencia. Estos datos se pueden obtener de manera teórica, por hipótesis o comparación con otros modelos, o de manera experimental. Una vez caracterizado el modelo se procede a su resolución. Los datos que se obtengan de esta resolución se pueden utilizar para desarrollar versiones posteriores del sistema.

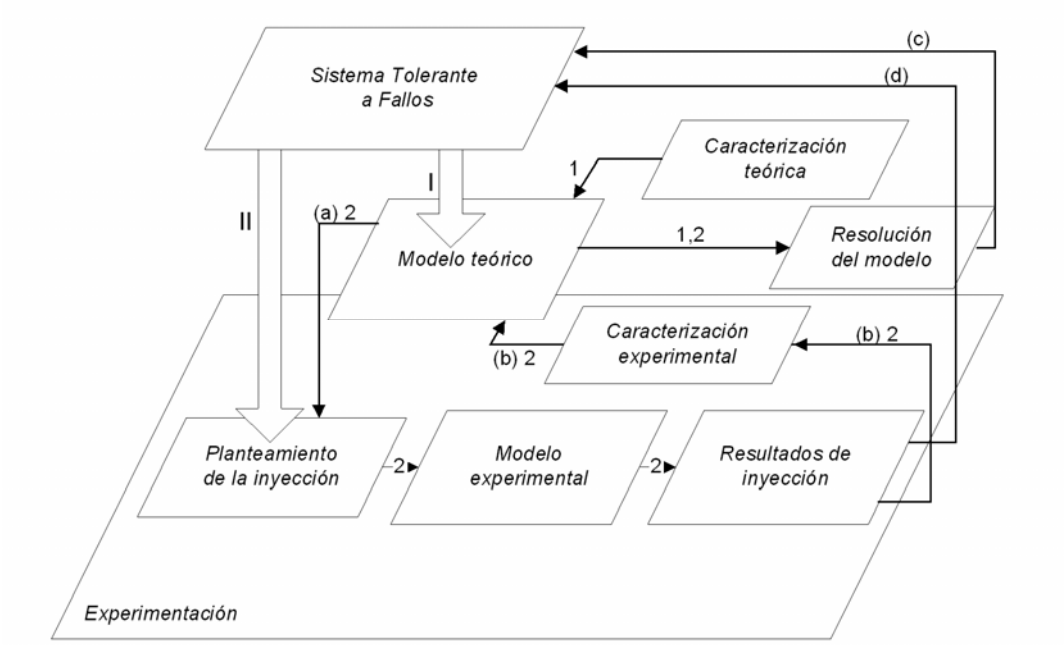


Figura 2.12: Diagrama de bloques del proceso de validación teórica y experimental mediante inyección de fallos

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

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

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

2.10 RESUMEN Y CONCLUSIONES

En este capítulo se ha definido la confiabilidad, que es la propiedad de un sistema informático que permite depositar una confianza justificada en el servicio que proporciona. La confiabilidad se debe ver desde el punto de vista de sus atributos, impedimentos y medios.

Los atributos de la confiabilidad permiten expresar las propiedades que se esperan de un sistema así como valorar la calidad del servicio entregado. Los atributos de la confiabilidad son disponibilidad, fiabilidad, seguridad-inocuidad, confidencialidad, integridad y mantenibilidad.

Los impedimentos son circunstancias no deseadas que provocan la pérdida de la confiabilidad. Los impedimentos de la confiabilidad son fallos, errores y averías. 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.

Los medios para conseguir la confiabilidad son los métodos y técnicas que capacitan al sistema para entregar un servicio en el que se pueda confiar, así como permiten al usuario tener confianza en esa capacidad. Los medios son la prevención de fallos, la tolerancia a fallos, eliminación de fallos y predicción de fallos. La prevención de fallos se corresponde con las técnicas generales de diseño de sistemas. La tolerancia a fallos consiste en la utilización de técnicas que permitan al sistema cumplir con su función a pesar de la existencia de fallos. La eliminación de fallos intenta, mediante las etapas de verificación, diagnóstico y corrección, reducir la existencia de fallos en el sistema. La predicción de fallos intenta estimar el número de fallos y su gravedad en un sistema.

La eliminación de fallos está muy ligada a la predicción. Al conjunto de las dos se le denomina validación, que puede ser teórica o experimental según si se aplica sobre un modelo del sistema o sobre un prototipo. La validación experimental permite calcular los valores de parámetros como las latencias y los coeficientes de cobertura de detección de errores de una manera más sencilla que con los métodos analíticos. Se puede realizar mediante inyección de fallos, esto es, introduciendo deliberadamente fallos en un modelo o prototipo del sistema.

Capítulo 3

TÉCNICAS DE INYECCIÓN DE FALLOS

3.1 INTRODUCCIÓN

Vista la utilidad de la inyección de fallos como herramienta de validación de la confiabilidad, a continuación se van a revisar cuáles son las técnicas de inyección de fallos que se han utilizado hasta la fecha.

La clasificación más aceptada de las técnicas de inyección de fallos [Hsueh97] [Saiz06] [Yu0l] las divide en tres grandes grupos, como se puede ver en la figura 3.1:

- La inyección de fallos basada en modelos utilizando para ello la simulación, también llamada como SBFI (del inglés "*Simulation-Based Fault Injection"*), que se aplica sobre un modelo simulado del sistema. Generalmente este modelo está realizado en un lenguaje de descripción del hardware del que se simula su comportamiento. O mediante la emulación, donde a partir de un prototipo hardware del sistema bajo estudio, se puede realizar una validación de su comportamiento en fases tempranas de diseño del mismo.

- La inyección de fallos física, también llamada implementada mediante hardware, o HWIFI (del inglés *"HardWare Implemented Fault Injection"*) inyecta fallos en el sistema real o en un prototipo del mismo utilizando mecanismos físicos.

- La inyección de fallos software, también llamada SWIFI (del inglés "*SoftWare Implemented Fault Injection"*) utiliza mecanismos software para inyectar los fallos. Los fallos pueden inyectarse previamente a la ejecución del programa o aplicarse sobre el sistema en ejecución. Frente a las dos anteriores tiene la ventaja de poder emular tanto fallos de tipo hardware como de tipo software.

Las técnicas existentes de inyección de fallos se pueden clasificar en una primera aproximación según la fase de diseño del sistema al que se aplican. Así, en las primeras etapas de diseño de un sistema se puede utilizar la inyección de fallos basada en simulación o emulación sobre modelos del sistema, mientras que en las etapas más avanzadas, se puede hacer inyección de fallos sobre un prototipo del sistema. Dentro de la inyección sobre prototipos se

pueden considerar la inyección física de fallos y la inyección de fallos por software. En los siguientes apartados se analizan cada una de estas técnicas.

3.2 INYECCIÓN DE FALLOS BASADA EN MODELOS

El objetivo de la inyección de fallos basada en modelos a través de la simulación o emulación es comprobar, en una etapa temprana del proceso de diseño, si el comportamiento en presencia de fallos de un sistema en desarrollo coincide con la especificación. El sistema puede ser un circuito integrado, una unidad funcional o un subsistema del sistema completo. El primer requisito, en el caso de la simulación, es disponer de un modelo del sistema, normalmente un modelo en VHDL (del inglés "*Very High Speed Integrated Circuit Hardware Description Language*"). En el caso de la emulación las actuales FPGAs (del inglés "*Field-Programmable Gate Array*") permiten obtener un prototipo hardware del sistema a evaluar. A partir de estos modelos se puede simular su comportamiento tanto en funcionamiento normal como en presencia de fallos. La simulación normalmente se basa en la utilización de herramientas EDA (del inglés "*Electronic Design Automation*") a las que se añade alguna herramienta especializada en la inyección de fallos. Actualmente esta técnica se utiliza también en predicción de fallos. Esto es posible debido a la existencia de herramientas especiales para inyectar grandes cantidades de fallos de manera aleatoria. En el caso de la emulación las FPGAs permiten implementar modelos en un lenguaje de descripción hardware (HDL del inglés "*Hardware Description Language*") de sistemas VLSI (del inglés "*Very Large Scale Integration*"), dichos dispositivos ofrecen unas posibilidades de reconfiguración muy interesantes a la hora de emular el comportamiento de un sistema en presencia de fallos.

Una de las diferencias más importantes entre la simulación y la emulación radica en la forma en la que se inyectan los fallos. En la simulación la inyección de fallos es llevada a cabo alterando los valores lógicos de los elementos del modelo (señales, variables, etc.). En este caso como se trata de una simulación software, para cambiar el valor de dichos elementos se hace uso de fragmentos de código (saboteadores, mutantes) o comandos de un simulador. Por el contrario, en la emulación de fallos se dispone de un dispositivo FPGA que funcionalmente se comporta igual que el sistema que se desea validar. En este caso se hace uso de las capacidades de reconfiguración que disponen dichos dispositivos para emular la ocurrencia de fallos en los circuitos implementados.

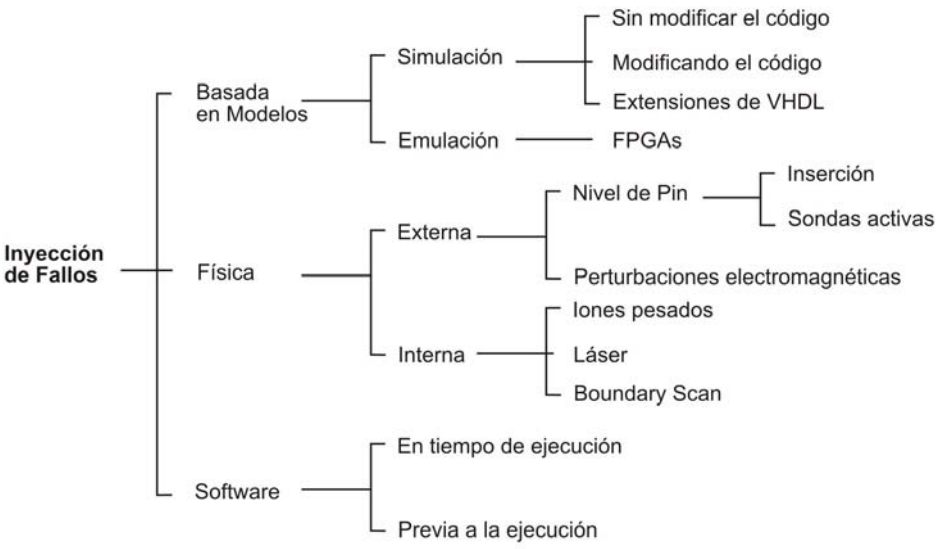


Figura 3.1: Clasificación de las técnicas de inyección de fallos

La inyección de fallos basada en simulación se puede utilizar a diferentes niveles de abstracción del sistema, desde el nivel de circuito integrado al nivel de sistema en un sistema distribuido. El sistema más utilizado hasta ahora ha sido el nivel de circuito integrado.

La tabla 3.1 [Vigneron97] muestra un resumen de las herramientas comerciales de simulación asociadas a entornos EDA. Los fallos que se inyectan suelen ser de los siguientes tipos: Pegado o *stuck-at*, línea abierta, retardo, inversión o *bit-flip*, cortocircuito y puente entre conexiones. Según sus características temporales los fallos pueden ser permanentes, transitorios o intermitentes.

Un objetivo que se busca en el proceso de diseño de sistemas con alta confiabilidad es acoplar las tareas de diseño y de validación basada en inyección de fallos. De esta manera es posible desarrollar el proceso de diseño en pasos sucesivos para conseguir optimizar las elecciones de diseño y la implementación de las acciones correctivas necesarias. Para conseguir este objetivo de obtener métodos de diseño coherentes e integrados, hay que tener en cuenta la necesidad de utilizar los diferentes lenguajes de descripción de hardware. En este sentido, el lenguaje que más se ha utilizado hasta ahora ha sido el VHDL, que presenta las siguientes características:

- Posibilidad de describir tanto la estructura como el comportamiento del sistema en un sólo elemento sintáctico.
- Muy utilizado actualmente para diseño digital.
- Capacidad de realizar descripciones jerárquicas a diferentes niveles de abstracción [Dewey92] [Aylor92].
- Buenas prestaciones de modelado a alto nivel de sistemas digitales.

Tabla 3.1: Herramientas EDA y simuladores comerciales

Producto	Distribuidor	Simulador	Nivel de Abstracción	Lenguaje
Verilog XL	Cadence	Verilog XL	Comportamental, puertas	Verilog
Voyager	Ikos	Voyager VS Voyager CS Voyager CSX	Comportamental Comportamental, puertas Comportamental, puertas con acelerador	VHDL
Idea Station Entry Station Quick HDL Pro	Mentor Graphics	Quicksim Quick HDL Quick HDL Quicksim	Puertas Comportamental, puertas Comportamental, puertas Puertas	BLM VHDL, Verilog VHDL, Verilog BLM
Silos	Simucad (Intsys)	Silos	Comporamental, puertas, transistores	Verilog, Verilog-A
Visual HDL	Summit (Backstreet Intsys)	Visual HDL Visual Verilog	Sistema, comportamental, puertas Sistema, comportamental, puertas	VHDL Verilog
Vulcan	Veda	Vulcan	Comportamental, puertas	VHDL
Fusion HDL	Viewlogic	Speedwave VCS Viewsim	Comportamental Comportamental, puertas Puertas	VHDL Verilog
Paradigm	Zycad	Paradigm VIP	Puertas	VHDL,Verilog
PSPICE	Microsim (ALS Design)	PSPICE	Puertas, transistores	ABM

Las técnicas de inyección de fallos basada en simulación se pueden clasificar en tres grandes grupos: Inyección de fallos sin modificación del código VHDL, inyección de fallos modificándolo e inyección de fallos con modificaciones del lenguaje:

- Sin modificar el código VHDL, la inyección se realiza utilizando órdenes del simulador, incluyéndolas a la hora de compilar el código VHDL.
- Existen dos maneras de modificar el código VHDL para inyectar fallos. La primera se basa en añadir componentes específicos de inyección de fallos, que se llaman saboteadores. La segunda se basa en modificar (o mutar) componentes ya existentes del código VHDL, lo que genera descripciones de componentes modificadas, llamadas mutantes.
- Por último, han aparecido algunas técnicas variantes de las anteriores. Estas técnicas se basan en extensiones del lenguaje VHDL orientadas a facilitar la simulación de los fallos.

Ejemplos de entornos de inyección de fallos basada en VHDL son [Gi197b] [Gil98] desarrollado por el GSTF (Universidad Politécnica da Valencia), MEFISTO-L [Boue98] del LAAS de Tolouse (Francia) y MEFISTO-C [Folkesson98] de la Chalmers University of Technology en Göteborg (Suecia). Como ejemplo de herramienta basada en extensiones del lenguaje VHDL cabe citar VERIFY (VHDL-*based Evaluation of Reliability by Injecting Faults*) [Sieh97]. En esta herramienta se amplía el lenguaje VHDL de manera que se puede describir el comportamiento de componentes hardware en caso de fallos con la señal de inyección de fallos junto con su tasa de ocurrencia.

Ventajas de la inyección de fallos basada en simulación:

- Posibilidad de inyectar fallos antes de construir el prototipo.
- Alcanzabilidad dependiente del nivel de detalle del modelo.
- Controlabilidad dependiente del nivel de detalle del modelo.
- Observabilidad dependiente del nivel de detalle del modelo.
- Buena reproducibilidad de los fallos.
- No se puede dañar ni afectar de ninguna manera al comportamiento del sistema bajo prueba.
- El coste de la infraestructura necesaria es relativamente bajo (siempre que haya un ordenador con las herramientas necesarias)
- Se pueden medir fácilmente latencias de detección del error o de recuperación.

Inconvenientes de la inyección de fallos basada en simulación:

- Tiempos de simulación altos, lo que hace que el proceso de inyección sea muy lento en ordenadores medios.
- Exactitud de los resultados dependiente de la bondad del modelo ejecutado. Está claro que cuanto mejor sea el modelo habrá mejor exactitud y mayores tiempos de ejecución. Esto es debido al alto nivel de detalle de un modelo estructural.
- No existe la posibilidad de inyectar fallos en un prototipo en tiempo real.

Por otro lado la inyección de fallos basada en emulación tiene como ventajas principales:

- Al tener un prototipo hardware del sistema a validar es mucho más rápido poner en marcha un modelo del sistema que en la simulación.
- Además es posible conectar el prototipo del sistema al entorno sobre el que se está trabajando.
- Frente a la HWIFI se tiene una mejor accesibilidad y un mayor conjunto de modelos de fallos que permite un análisis más preciso de la patología de los mismos.

Por el contrario las desventajas de esta técnica son:

- No todos los modelos HDL pueden ser implementados en una FPGA. En este caso se recurre a utilizar un modelo más sintetizado del sistema.
- No es posible inyectar fallos en todas las partes del modelo, teniendo en este caso menos posibilidades que la simulación.
- Actualmente hay un conjunto limitado de vendedores de emuladores.
- Dependiendo del emulador utilizado se puede llegar a tener una lectura de datos muy restringida.
- Es necesario reconfigurar la FPGA y en ocasiones descargar el fichero de configuración.
- Y debido que la tecnología cambia tan rápidamente, los nuevos modelos de FPGAs pueden hacer que trabajos previos queden obsoletos.

Actualmente, en esta línea de investigación dentro del Grupo de Sistemas Tolerantes a Fallos (GSTF) de la Universidad Politécnica de Valencia, se ha planteado la necesidad de combinar la simulación y la emulación como técnicas de inyección de fallos que se espera permitan la definición de una metodología de validación rápida de modelos para sistemas VLSI en fases tempranas del diseño de los mismos [Saiz06].

3.3 INYECCIÓN FÍSICA DE FALLOS

En este caso se intenta acercar más a la realidad los fallos que se introducen en el sistema. Todas las técnicas de inyección física de fallos realizan una inyección real del fallo en los terminales de los circuitos o emulan sus consecuencias (errores) por medio de perturbaciones externas o internas. Hay dos tipos de técnicas de inyección de fallos, internas y externas. En las técnicas de inyección externa los fallos se inyectan fuera de los componentes a validar, por ejemplo, en los pines de los circuitos integrados. En las técnicas internas los fallos se inyectan dentro del componente a validar, por ejemplo, mediante láser, radiaciones de iones pesados o con mecanismos especiales integrados en el hardware del sistema.

3.3.1 Inyección física externa

Hasta la fecha se han utilizado dos técnicas de inyección de fallos externa, que son la que se corresponde con la inyección a nivel de pin de circuito integrado [Arlat90] [Madeira94] [Gi197a] e inyección mediante perturbaciones electromagnéticas [Damm88] [Karlsson95].

3.3.1.1 Inyección física a nivel de pin

Para llevar a cabo la inyección física de fallos a nivel de pin, se utiliza una herramienta especial que modifica los valores lógicos en los pines de los circuitos integrados o en las líneas de comunicación y control. Estas modificaciones de los valores lógicos deben ser representativas de los fallos que aparecen en la fase operacional de un sistema.

Los fallos se localizan en aquellos pines o conexiones del sistema en los que se modifica el valor lógico. Estos pines pueden ser de cualquier componente del sistema, aunque según cuál sea el objetivo de la campaña de inyección se pueden restringir a un determinado subconjunto en caso de eliminación de fallos o se pueden elegir aleatoriamente en caso de predicción de fallos.

Los modelos de fallos físicos más utilizados son el pegado, puente y línea abierta. El modelo de fallo lógico dependerá de cuál sea la condición que se utilice para disparar la inyección, pero está directamente relacionado con la corrupción de la información de comunicación interna y la corrupción de elementos funcionales y de control. Indirectamente se puede realizar una corrupción de información de almacenamiento corrompiendo las instrucciones que envíen datos a través de los buses o la red de comunicaciones. En este caso los fallos pueden ser permanentes, transitorios o intermitentes.

Existen dos tipos de técnicas de inyección física a nivel de pin:

- Técnica de **Sondas activas**: Cuando se utiliza esta técnica se inyecta el fallo directamente en los terminales de los circuitos integrados, conectores o pistas de circuito impreso, sin desconectar ningún componente. La sonda del inyector de fallos fuerza un cero o uno lógicos en los puntos elegidos.

- Técnica de **Inserción**: Se coloca un dispositivo especial en el lugar del componente donde se va a inyectar y entre el mismo y su soporte (normalmente un zócalo o conector). Este dispositivo intermedio es el encargado de inyectar los fallos. Antes de inyectar el fallo se corta la conexión entre el circuito y el sistema. De esta forma la inyección se realiza sobre una línea en alta impedancia y, por lo tanto, al no haber forzado de ninguna señal no existe la posibilidad de dañar el componente sobre el que se inyecta.

Algunas herramientas de inyección física a nivel de pin son las siguientes:

- **MESSALINE** desarrollado en el LAAS de Toulouse (Francia) [Arlat90]. Podía inyectar fallos múltiples en cualquier momento, permitiendo además la automatización de los experimentos. Este inyector de fallos implementaba tanto la técnica de sondas activas como la de inserción de zócalo.
- **RIFLE**, de la Universidad de Coimbra (Portugal) [Madeira94]. Este inyector tenía un circuito de memoria de traza que permitía acelerar los experimentos de inyección. En este caso sólo se utilizaba la técnica de inserción de zócalo.
- **AFIT** (“*Advanced Fault Injection Tool*”), desarrollado por el Grupo de Sistemas Tolerantes a Fallos de la Universidad Politécnica de Valencia [Gi197a] permite inyectar fallos en sistemas de alta velocidad. En este caso sólo se implementa la técnica de sondas activas, dado que el objetivo es inyectar fallos en componentes de alta velocidad que normalmente utilizan encapsulados de montaje superficial.

Ventajas de la inyección de fallos a nivel de pin:

- Al inyectar sobre el sistema real, los resultados obtenidos se consideran representativos de los reales.
- Dado que las herramientas son externas al sistema no se sobrecarga la aplicación bajo prueba.
- La inyección se puede activar por un disparo temporal o espacial, y se efectúa la inyección sin parar el sistema.
- El retardo por inyección no es muy alto si se utiliza una herramienta automática para gestionarlo.
- Los fallos son fácilmente reproducibles.
- Esta técnica tiene una buena controlabilidad espacial dado que los fallos se inyectan en lugares bien especificados, y algo de menor controlabilidad temporal dado que el instante de inyección puede ser más difícil de controlar en sistemas de alta velocidad.
- Es fácil medir latencias de detección y recuperación.

Inconvenientes de la inyección de fallos a nivel de pin:

- Alcanzabilidad restringida, sobre todo en las últimas generaciones de chips VLSI y sistemas en chip o SoC's (del inglés "*Systems on a Chip*") que tienen una relación complejidad/número de pines muy alta.
- Baja observabilidad, que depende de la complejidad de los circuitos integrados y de si en el diseño se ha tenido en cuenta la orientación al test.
- Si se usa la técnica de sondas activas existe una baja probabilidad de dañar el circuito integrado. Con la técnica de inserción de zócalo esta probabilidad es prácticamente nula.
- Dificultad de realizar el cableado entre la herramienta de inyección y el sistema, sobre todo si se utilizan en el sistema componentes de montaje superficial. Además, para este tipo de componentes es prácticamente imposible utilizar la técnica de inserción de zócalo.
- Coste alto de la infraestructura necesaria. Es necesario una herramienta especial de inyección de fallos.
- Interferencias de la herramienta de inyección sobre el sistema, especialmente si se trata de sistemas con frecuencias de reloj elevadas.

3.3.1.2 Inyección física por interferencias electromagnéticas

El trabajo más importante es el realizado en la Technical University of Vienna (Austria) [Damm88][Karlsson95]. Su sistema de inyección de fallos utiliza los equipos de test sobre inmunidad electromagnética según el estándar EN 61000-4-4. Este equipo genera ráfagas de interferencias con las siguientes características temporales: duración 15ms, frecuencias de 1.25, 2.5, 5 ó 10 KHz, y tensiones que se pueden elegir desde 225V hasta 4400V. Este tipo de pruebas se realizan normalmente para verificar el correcto funcionamiento de un equipo completo ante interferencias electromagnéticas, en este caso se eliminan los mecanismos de protección electromagnética para aumentar la efectividad de los mismos. Los fallos resultan de las interferencias que se localizan en las pistas del circuito impreso y entran en los circuitos integrados del sistema a validar. El modelo físico de estos fallos es el de perturbación electromagnética.

Ventajas de la inyección de fallos por interferencias electromagnéticas:

- Al inyectar sobre el sistema real, los resultados obtenidos se consideran representativos de los reales.
- Dado que las herramientas son externas al sistema no se sobrecarga la aplicación bajo prueba.
- La duración de una campaña de inyección no es muy alta si se gestiona mediante un sistema automático. Aunque para comprobar si el fallo es efectivo se deben ir comparando los resultados con los obtenidos de un sistema libre de fallos, lo que ralentiza el proceso.
- Fácil de aplicar a un prototipo ya que no hay contacto físico entre el sistema y la herramienta de inyección.
- Baja probabilidad de dañar al prototipo.

Inconvenientes de la inyección de fallos por interferencias electromagnéticas:

- Alcanzabilidad restringida, al igual que en el caso anterior, ya que las interferencias se introducen a nivel de pin.
- Baja controlabilidad espacial y temporal.

- Baja observabilidad. Depende de la complejidad de los circuitos integrados y de si el diseño del sistema está orientado al test. Se debe tener en cuenta que los síndromes de error se detectan por comparación con un sistema libre de fallos, lo que en la bibliografía inglesa se denomina como *golden-unit*.
- Las interferencias pueden provocar fallos múltiples, por ejemplo inducidos a través de las líneas de alimentación y masa del sistema. Este tipo de fallos es muy difícil de detectar.
- Los experimentos son difíciles de reproducir.
- Hay un alto coste en infraestructura. Aunque se utiliza una herramienta disponible comercialmente.
- Las latencias son difíciles de medir.

3.3.2 Inyección física interna

Denominamos inyección física interna a la que utiliza mecanismos físicos para introducir el fallo directamente en el interior de los componentes. Actualmente se utilizan tres técnicas de inyección física interna: Mediante iones pesados, mediante técnicas láser y mediante *Boundary Scan*.

3.3.2.1 Inyección física por radiación de iones pesados

Esta técnica se desarrolló en la Chalmers University of Technology, en Göteborg (Suecia), [Gunneflo89] [Karlsson95]. En esta técnica se generan fallos del tipo inversión o "*bit-flip*" dentro del componente irradiado. Se utiliza una fuente radiactiva de Californio 252. En este caso lo que se intenta es simular fallos transitorios producidos por fuentes externas, como por ejemplo las interferencias radiactivas, eléctricas o electromagnéticas. Para conseguirlo se debe eliminar la tapa del encapsulado del componente a irradiar e introducir todo el conjunto componente y fuente radiactiva dentro de una cámara de vacío dado que las partículas del aire podrían modificar la trayectoria e incluso parar los iones.

La característica más interesante de esta técnica es la posibilidad de introducir perturbaciones en toda la superficie de silicio del circuito integrado. De esta forma se pueden inyectar fallos en zonas del chip inaccesibles para otras técnicas.

Ventajas de la inyección de fallos por radiación de iones pesados:

- Al inyectar sobre el sistema real, los resultados obtenidos se consideran representativos de los reales.
- Dado que las herramientas son externas al sistema no se sobrecarga la aplicación bajo prueba.
- La duración de una campaña de inyección no es muy alta si se gestiona mediante un sistema automático. Aunque para comprobar si el fallo es efectivo se deben ir comparando los resultados con los obtenidos de un sistema libre de fallos, lo que ralentiza el proceso.
- Buena alcanzabilidad, ya que los fallos se pueden introducir en cualquier punto del sistema.
- Poca interferencia sobre los componentes del sistema que no se irradian.

Inconvenientes de la inyección de fallos por radiación de iones pesados:

- Los experimentos son difíciles de preparar. Hace falta eliminar la tapa del encapsulado e introducirlo en una cámara al vacío. Los componentes se degradan durante el proceso de inyección.
- Probabilidad de dañar permanentemente el componente, especialmente si se trata de un componente CMOS debido al fenómeno conocido como "*Latch-up*".
- Baja reproducibilidad de los experimentos, ya que los fallos se inyectan en posiciones desconocidas.
- Baja controlabilidad espacial y ninguna controlabilidad temporal, ya que el proceso de generación de iones pesados es aleatorio.
- Baja observabilidad, que depende de la complejidad del circuito sobre el que se inyecte.
- Hay un alto coste en infraestructura.

3.3.2.2 Inyección física por radiación láser

En esta técnica [Sampson98] se inyectan fallos transitorios en puntos muy bien controlados dentro de un circuito integrado sin encapsulado. La inyección se efectúa apuntando con un haz láser sobre la superficie del silicio. Este láser genera pares electrón-hueco en el semiconductor de la misma manera que los iones pesados utilizados en la técnica anterior.

Ventajas de la inyección de fallos por radiación láser:

- Al inyectar sobre el sistema real, los resultados obtenidos se consideran representativos de los reales.
- Dado que las herramientas son externas al sistema no se sobrecarga la aplicación bajo prueba.
- La duración de una campaña de inyección no es muy alta si se gestiona mediante un sistema automático. Aunque para comprobar si el fallo es efectivo se deben ir comparando los resultados con los obtenidos de un sistema libre de fallos, lo que ralentiza el proceso.
- Buena alcanzabilidad, ya que los fallos se pueden introducir en cualquier punto del sistema.
- Alta controlabilidad espacial, ya que el punto de inyección se puede determinar con exactitud.
- Mejor reproducibilidad de los experimentos que con iones pesados, ya que el láser se puede volver a apuntar al mismo lugar con bastante precisión.

Inconvenientes de la inyección de fallos por radiación láser:

- Los experimentos son difíciles de preparar. Hace falta eliminar la tapa del encapsulado. Como en el caso anterior, los componentes se degradan durante el proceso de inyección.
- Baja observabilidad, que depende de la complejidad del circuito sobre el que se inyecte.
- Baja controlabilidad temporal.
- Hay un alto coste en infraestructura.

3.3.2.3 Inyección mediante Boundary Scan

Esta técnica se encuentra a caballo entre las internas y las externas, ya que si bien el mecanismo que se utiliza para inyectar el fallo es interno al sistema, el fallo que se emula es una alteración del estado lógico de los pines del circuito integrado, como en las técnicas externas.

En la bibliografía se le denomina inyección basada en *Boundary Scan* o inyección basada en cadenas de exploración (*scan-chain*). Se basa en la utilización del estándar conocido con los nombres JETAG, JTAG, Boundary Scan o IEEE 1149.1 para acceder a los registros denominados BSC (*Boundary Scan Cells*). Estos registros están situados en cada entrada o salida digital del circuito integrado. En funcionamiento normal son los encargados de transferir los niveles lógicos entre el circuito y los pines y en modo test se pueden reprogramar para transferir otros datos diferentes. En esta técnica se utiliza este modo de test para modificar artificialmente estos niveles emulando un fallo externo.

Esta técnica se desarrolló en la universidad de Chalmers [Folkesson98] con la herramienta FIMBUL (*Fault Injection and Monitoring using BUilt in Logic*), integrada más tarde en GOOFI (*Generic Object-Oriented Fault Injector*) [Aidemark01][Aidemark03]. En [Santos03] el autor estudia esta técnica para integrarla en Xception [Carreira95], que actualmente es una herramienta de inyección de fallos basada en software. En ese mismo trabajo se proponen cambios en el estándar Boundary Scan para mejorar su pobre comportamiento temporal de cara a mejorar la inyección.

Ventajas de la inyección de fallos mediante Boundary Scan:

- Al inyectar sobre el sistema real, los resultados obtenidos se consideran representativos de los reales.
- Alcanzabilidad restringida a las entradas/salidas del circuito.
- Buena observabilidad, ya que los mecanismos de Boundary Scan se diseñaron precisamente para test.
- Buena reproducibilidad.
- No se puede dañar al prototipo.
- Buena portabilidad, ya que utiliza un estándar utilizado en test.

Inconvenientes de la inyección de fallos mediante Boundary Scan:

- En cada acceso hay que parar el sistema. Esto produce una ralentización muy grande del sistema, que puede resultar inaceptable para validar sistemas de tiempo real.
- Los experimentos tardan mucho en ejecutarse.
- Está limitada a sistemas que implementen el estándar Boundary Scan.

3.4 INYECCIÓN DE FALLOS POR SOFTWARE

El objetivo de la inyección de fallos por software es reproducir, a nivel lógico, los errores que se producen tras fallos en el hardware [Iyer95]. Aunque cada vez más, hay más trabajos que también apuntan la posibilidad de reproducir errores que ocurren en un sistema debidos a fallos de diseño del software (o fallos lógicos) [Madeira00][Duraes03].

La principal característica de este tipo de inyección es que es capaz de inyectar fallos en cualquier unidad funcional direccionable mediante el software, tales como memoria, registros, periféricos e incluso los controladores de los diferentes relojes del sistema. Por otro lado está la limitación de algunos registros internos a los que no se tiene acceso [Gi102][Iyer95].

Los modelos físicos de fallo son la inversión o *bit-flip* y el pegado o *stuck-at*. A nivel lógico puede ser corrupción de información almacenada (al inyectar sobre registros o memoria), corrupción de comunicaciones (al inyectar sobre los mensajes) y corrupción de unidades

funcionales (al inyectar sobre los registros de datos y/o control de las mismas). Además desde el punto de vista temporal se pueden inyectar fallos de tipo permanente, transitorio e intermitente.

Respecto a la portabilidad de las herramientas, un inyector puede ser un conjunto de rutinas que se ejecutan junto con la aplicación [Kanawati95][Han95][Rodríguez99] aunque también existen herramientas de inyección de fallos software que no ejecutan código en el sistema a inyectar sino que modifican el código previamente a la ejecución y observan [Kao93][Han95]. En ambos casos, una herramienta de inyección software de fallos suele estar diseñada específicamente para un sistema concreto y su utilización restringirse al mismo.

Precisamente por este mismo motivo, la propia naturaleza de esta técnica hace que sea muy factible su implementación desde un punto de vista económico, y ha propiciado que hayan aparecido muchas herramientas basadas en inyección software de fallos. A continuación comentamos las más significativas.

FIAT

FIAT (*Fault Injection Based Automated Testing Environment*) [Segall88] fue una de las primeras herramientas de inyección de fallos por software. En esta herramienta se modificaba la imagen de memoria de la aplicación. Se inyectan en el software en ejecución patrones de errores que son representativos de los errores que con más probabilidad producirían los fallos en el software y en el hardware. El objetivo de esta herramienta es encontrar las deficiencias de los mecanismos de detección y recuperación de errores de un sistema y comparar mecanismos alternativos, evaluando cuantitativamente su efectividad relativa.

EFA

EFA (*Experimental environment for Fault tolerance Algoritms*) [Echte92] fue desarrollado en la Universidad de Dortmund para probar las características de tolerancia a fallos de diferentes algoritmos. Se utiliza en un sistema distribuido, en el que cada nodo tiene su propio inyector de fallos. El inyector se intercala en la capa de enlace de datos del sistema de comunicaciones y corrompe los mensajes que se transmiten para estudiar el comportamiento del sistema frente a errores de comunicaciones. El comportamiento del sistema se define a partir de los mensajes que transmite.

DOCTOR

DOCTOR (*integrateD sOftware implemented fault injeCTiOn enviRon*ment) [Han95] desarrollado en la Universidad de Michigan inyecta fallos permanentes en el procesador cambiando instrucciones de la imagen de memoria por otras. DOCTOR puede inyectar fallos transitorios en el procesador, memoria y en la interfaz de comunicaciones del sistema distribuido HARTS. Todos estos tipos de fallos se pueden inyectar mediante el uso de un agente de inyección que se ejecuta de manera concurrente en el sistema sobre el que se van a realizar los experimentos.

FERRARI

En FERRARI (*Fault and ERRor Automatic Real-time Injector*) [Kanawati95] el inyector se ejecuta de forma concurrente con la aplicación. En este caso se utiliza la función "*Ptrace*" de UNIX antes de lanzar a ejecución la aplicación. De esta forma el código se ejecuta en modo traza y el inyector tiene acceso a su mapa de memoria. Gracias a esta característica se puede corromper la imagen en memoria del proceso insertando instrucciones erróneas en las posiciones donde se debe activar un fallo. Un detalle interesante es que aunque el nombre de la herramienta incluye las palabras real-time en realidad la aplicación bajo prueba no se ejecuta a su velocidad normal debido a estar en modo traza y a la interferencia del programa inyector.

FINE/DEFINE

FINE [Kao93]/DEFINE [Kao94] son las versiones monoprocesador y distribuida de la misma herramienta. Como en los casos anteriores se utilizan mecanismos de depuración para

efectuar la inyección, aunque en este caso se modifica el manejador de interrupción del reloj para inyectar fallos de CPU y bus. Esta herramienta está orientada al estudio de la propagación de los fallos dentro del sistema operativo. Por este motivo está muy integrada con el mismo, de manera que se puede considerar como una capa entre el núcleo del sistema operativo y el hardware.

Xception

Xception [Carreira95][Carreira98], desarrollado en la Universidad de Coimbra, utiliza para planificar el instante de inyección la excepción prevista en el procesador para dar servicios de tiempo al sistema operativo. Esta herramienta está basada en la utilización de los mecanismos de depuración implementados en el procesador a través de las excepciones para realizar la inyección. Primero se deja ejecutar a la aplicación hasta el momento del disparo, en el que se entra en modo traza y se ejecuta paso a paso hasta la instrucción a inyectar. Una vez en la instrucción a inyectar se decodifica y se decide qué parámetros de la misma modificar en función del tipo de fallo que se pretenda inyectar.

MAFALDA

Otra herramienta que utiliza los mecanismos de depuración del procesador es MAFALDA (*Microkernel Assessment by Fault injection AnaLysis and Design Aid)* [Rodríguez99], desarrollada en el LAAS-CNRS en Toulouse. Mafalda está orientada a la validación de microkernels [Fabre00] y permite realizar dos tipos de inyecciones: La corrupción de los parámetros de entrada al llamar a una primitiva del microkernel y la inyección de fallos en segmentos de memoria de código y de datos. En el caso de la inyección en los parámetros de las llamadas al sistema se interceptan las llamadas, bien mediante una librería de llamadas modificada para inyección o bien mediante un *breakpoint* en la entrada al microkernel haciendo que se ejecute antes de la llamada un manejador encargado de corromper los parámetros. En el caso de la inyección en memoria se definen fallos permanentes o transitorios disparados por eventos espaciales o temporales. En caso de los transitorios, tras la inyección se entra en modo traza, se avanza en la ejecución paso a paso y se restaura la memoria. Para el estudio de sistemas de tiempo real también han desarrollado una herramienta denominada como MAFALDA-RT, en este caso para llevar a cabo la inyección de fallos se para el reloj del sistema con el objetivo de que el propio proceso de inyección de fallos no consuma tiempo de ejecución del sistema, pero como veremos posteriormente esta solución que virtualiza el tiempo no es muy aconsejable para la evaluación/certificación de sistemas de tiempo real.

SOFI

SOFI (*SOftware Fault Injector*) [Campelo99a][Campelo99b] se desarrolló en el Grupo de Sistemas Tolerantes a Fallos de la Universidad Politécnica de Valencia. Está diseñado para validar nodos de proceso en sistemas empotrados distribuidos tolerantes a fallos. Un PC se encarga de supervisar el proceso de inyección y envía una señal de inicio de inyección al sistema bajo estudio. En éste, un agente de inyección reacciona a esta señal parando la ejecución de la aplicación y corrompiendo la memoria. Como característica destacable cabe señalar que logra una efectividad del 100% en el segmento de código, lo cual incide drásticamente en el tiempo total necesario para realizar una campaña de inyección de fallos. Como inconveniente señalar que la actuación de dichos agentes provoca una cierta intrusión en el sistema.

FlexFi

De FlexFi [Benso99a] se han creado tres implementaciones con diferente intrusión y coste: una puramente software que utiliza las excepciones del procesador, otra apoyada en un circuito hardware específico [Benso99b] para adquirir una traza de ejecución del sistema y acelerar los experimentos de inyección y una tercera que utiliza los mecanismos hardware de depuración implementados en los procesadores de Motorola (el puerto BDM) [Prinetto98]. Esta última implementación tiene la ventaja de no ser muy intrusiva en términos de inserción de código ajeno a la aplicación, sin embargo se experimenta una importante deceleración en la ejecución de la aplicación.

BALLISTA

Herramienta desarrollada en el Institute for Complex Engineered Systems en la Universidad del Carnegie Mellon [Kropp98], que tiene como objetivo el testeo automatizado de la robustez de módulos de software viendo su efectividad en el manejo de excepciones. Se centran principalmente en la corrupción de los parámetros de la API de sistemas operativos de propósito general que cumplen con la especificación POSIX. Para ello necesitan del SO bajo evaluación, de una descripción de las llamadas al sistema, parámetros y tipos de datos de los mismos. Con estos datos realizan una serie de tests de caja negra con el objetivo de identificar fallos reproducibles en el sistema que llevan al mismo a un estado de no respuesta o reseteado.

INERTE

INERTE (*Integrated NExus-based Real-Time fault injection tool for Embedded systems*) [Yuste03e] se desarrolló en el Grupo de Sistemas Tolerantes a Fallos de la Universidad Politécnica de Valencia. La herramienta se ha desarrollado con el fin de aportar la posibilidad de realizar una inyección de fallos hardware que no introduzca sobrecarga temporal sobre las zonas de memoria correspondientes a datos o código de sistemas empotrados de tiempo real con puerto de depuración Nexus™. La herramienta se ha utilizado para validar una aplicación de control inyectando fallos físicos.

3.5 COMPARACIÓN DE TÉCNICAS SOFTWARE DE INYECCIÓN DE FALLOS

De entre todas las técnicas, la primera clasificación es entre técnicas de inyección previa a la ejecución (***off-line***) y técnicas de inyección durante la ejecución (***runtime***). En las técnicas *off-line* se modifica el código de la aplicación previamente a la ejecución para insertar los fallos. Este tipo de técnicas presenta como principal ventaja una intrusión mínima en tiempo de ejecución, ya que no es necesario ejecutar ningún tipo de programa externo a la aplicación para realizar la inyección [Fuchs96]. Por este motivo estas técnicas se han utilizado para validar sistemas de tiempo real. Otra ventaja es la sencillez del método de inyección, ya que basta con modificar el código previamente a su ejecución. Actualmente, con interfaces que permiten sustituir el contenido de la memoria en tiempo de ejecución sin intrusión alguna, las antiguas limitaciones de estas técnicas en el modelo de fallos que eran capaces de inyectar se han superado. Los únicos fallos que se podían inyectar en memoria de código eran permanentes, ya que una vez la aplicación se encontraba en ejecución era imposible deshacer el cambio, actualmente con este tipo de interfaces esto se puede deshacer e inyectar fallos de tipo transitorio y permanente tanto en memoria de código como de datos.

En las técnicas denominadas *runtime* se inyectan los fallos mientras se está ejecutando la aplicación. Obviamente esto puede provocar una sobrecarga en el sistema sobre el que se está inyectando que viene tanto de la ejecución del código de inyección propiamente dicho como del que haya que ejecutar para llevar a cabo la monitorización del sistema. Algunas de las herramientas *runtime* se han diseñado sin tener en cuenta las restricciones temporales del sistema a evaluar. En otras sin embargo, se han diseñado los experimentos de manera que la sobrecarga de la herramienta en tiempo de ejecución se reduzca con el fin de poder inyectar fallos en sistemas de tiempo real [Cunha99][Rodríguez02]. En estos casos se tienen que crear funciones de inyección específicas para el sistema concreto que aprovechen todas las características del sistema para reducir al máximo el tiempo de ejecución, además se hace necesario evaluar cuál es la intrusión que necesariamente introducen para comprobar si los resultados que se pretenden obtener son o no válidos.

Tradicionalmente la ventaja de las técnicas *runtime* con respecto a las *off-line* es que se aumentaba el modelo de fallos que se podía inyectar. Por un lado se podían inyectar fallos transitorios en memoria de código, al poder modificar una posición de memoria de manera temporal, y fallos permanentes en memoria de datos, al poder reescribir el fallo cada vez que la aplicación lo eliminara. Por otro lado se podía controlar más el momento de disparo de la inyección, tanto si se quieren seguir criterios espaciales como temporales. Además se podían inyectar fallos de forma controlada en más lugares que con la técnica *off-line*, como por ejemplo en las llamadas al sistema operativo [Rodríguez99] o en las zonas de parámetros o de control de cualquier periférico del sistema.

Algunas de las herramientas de inyección de fallos permiten precisamente corromper la memoria del controlador de red para inyectar fallos de comunicaciones [Han95][Kao94]. Actualmente y como se ha comentado anteriormente este tipo de limitaciones, como son la alcanzabilidad, modelo de fallo e intrusión en el tiempo de ejecución, se han superado con la utilización de interfaces como Nexus™ que permiten la sustitución de memoria en tiempo de ejecución sin intrusión alguna. Esto ofrece la posibilidad que el contenido de una zona de memoria que fue sustituida en *off-line* sea posteriormente su contenido restituido en *runtime*, inyectando así tanto fallos de tipo permanente como transitorio.

3.5.1 Comparación de disparos

Un atributo que hay que tener en cuenta a la hora de llevar a cabo la inyección es la activación del fallo. Según las técnicas vistas, si se utiliza la técnica *off-line* el error se activa en el momento en que la ejecución pasa por el lugar donde el código se ha modificado, en caso de fallo en memoria de código, o usa el dato modificado si la inyección es en memoria de datos. Si se utiliza la técnica *runtime* se puede realizar también sincronizando la inyección con la ejecución del código o dependiendo de parámetros temporales.

Se tiene por tanto dos tipos de sincronización:

- **Temporal**: La inyección se activa dependiendo de parámetros temporales relativos a la ejecución de la carga en el sistema; por ejemplo que expire un temporizador desde el inicio de la aplicación.

- **Espacial**: La inyección se activa cuando la carga alcanza un estado o condición. Puede ser simplemente la ejecución de una instrucción concreta, como es en el caso de la inyección *off-line* o puede ser la activación de algún evento en el sistema, por ejemplo una excepción.

Siguiendo una sincronización temporal se puede hacer que aparezcan los fallos siguiendo un patrón temporal aleatorio lo que se supone más realista y conveniente para inyectar fallos transitorios e intermitentes. Sin embargo la sincronización espacial da resultados más repetitivos.

En el caso de inyección *runtime* además hay que tener en cuenta quién y dónde realiza el disparo. Pueden ser:

- **Temporizadores**: Realizando una sincronización temporal. Por ejemplo temporizadores internos al procesador sobre el que se va a inyectar el fallo o por ejemplo temporizadores de la herramienta externos al sistema. La ventaja de los primeros es la mayor sincronización con la aplicación, mientras que los segundos tienen a su favor el no consumir recursos del propio sistema [Benso99a].

- **Interrupciones condicionales**: Se genera una interrupción ante un evento del sistema. Si esta interrupción es interna puede provocar la activación del manejador asociado que realizará la inyección. En las herramientas hardware es la propia herramienta la que trata el evento. En este caso la sincronización es espacial [Carreira95].

- **Inserción**: En este caso se ejecuta, en el mismo sistema y de manera concurrente con la aplicación, un programa que la supervisa y dispara la inyección al reconocer un evento del sistema [Campelo99].

3.6 TRABAJOS DE INYECCIÓN DE FALLOS RELACIONADOS

Dentro de las técnicas y herramientas de inyección de fallos descritas hay una serie de trabajos que están más cercanos al objetivo de la presente tesis. Seguidamente se presentan cuáles son.

En primer lugar se debe considerar FlexFi [Benso99a], dado que hasta la fecha una de sus implementaciones [Rebaudengo99] ha sido una de las pocas herramientas que han utilizado los mecanismos de depuración incluidos en los microcontroladores actuales para controlar las campañas de inyección. En este caso se utiliza el puerto BDM (del inglés "*Background Debug Mode*") de los procesadores de Motorola. Este puerto permite comunicarse con los registros internos del procesador y realizar las acciones típicas de los emuladores, como cargar programas en memoria, ejecutarlos, poner puntos de ruptura, etc. El único inconveniente es la ralentización sobre el funcionamiento del sistema que supone el usar este modo de funcionamiento. A través del puerto BDM, esta herramienta puede inyectar fallos en la memoria del microcontrolador en tiempo de ejecución sin la necesidad de incluir junto con el programa código externo a la aplicación.

Tampoco se necesita utilizar ningún recurso genérico del procesador que podría necesitar la aplicación. Cuando se entra en modo BDM el procesador deja de ejecutar instrucciones de forma normal y la circuitería específica del procesador toma el control y ejecuta las instrucciones que van llegando a través del puerto BDM. Para ello se programa un punto de ruptura en una dirección concreta del programa para llevar a cabo la inyección de fallos. Tras ejecutar esa instrucción un número predeterminado de veces el programa se para automáticamente, se modifica la posición de memoria a inyectar y se vuelve a iniciar la ejecución. Al igual que en la mayoría de las herramientas software el modelo de fallo es el *bit-flip*. La aproximación que se utiliza en esta herramienta tiene dos inconvenientes fundamentales. El primero es que se produce una ralentización importante de la ejecución de la aplicación. El segundo es que sólo se puede utilizar en procesadores de Motorola que tengan un puerto BDM.

Aunque no está orientado a microcontroladores, Xception [Carreira98] fue la primera herramienta en utilizar los mecanismos de depuración incluidos en los procesadores actuales para llevar a cabo la inyección de fallos. Esta herramienta está orientada a la inyección en unidades funcionales del procesador, y por ello la forma en la cual se realiza la inyección es dependiente de la unidad funcional en la que se esté emulando el fallo. Por ejemplo, se ejecuta la aplicación hasta el instante de disparo del fallo, en ese momento se entra en modo traza. Se ejecuta la aplicación paso a paso hasta llegar a la instrucción sobre la que se va a inyectar el fallo, se decodifica y se elige el parámetro a modificar. En Xception se pueden inyectar fallos en todas las unidades funcionales del procesador, como por ejemplo la unidad de control de ejecución de instrucción, la unidad aritmética de enteros, la unidad aritmética de coma flotante, etc. También se pueden inyectar fallos en memoria. El disparo de la inyección puede ser una combinación de las siguientes condiciones: búsqueda de operando y carga desde una dirección específica, escritura en una dirección determinada y tiempo desde el inicio de la aplicación. Para

poder inyectar fallos en sistemas de tiempo real, en una versión de Xception, llamada RT-Xception [Cunha99] se reduce la intrusión hasta el mismo orden de magnitud que la latencia de la interrupción del núcleo de tiempo real y dentro de la dispersión en los tiempos de ejecución de la carga que se esté ejecutando.

Otra herramienta software relacionada es MAFALDA [Rodríguez99] que está orientada a la validación de núcleos de sistema operativo. MAFALDA inyecta fallos invirtiendo bits en los segmentos de código y datos de la memoria. Eligiendo estos bits cuidadosamente se emulan fallos hardware en la memoria o fallos software modificando los parámetros de entrada al núcleo. Al igual que Xception, MAFALDA utiliza los mecanismos de depuración del procesador para implementar la inyección. También al igual que Xception no está orientado a microcontroladores sino a procesadores de propósito general. Si nos fijamos en la intrusión temporal que provoca el uso de MAFALDA encontrarnos una versión [Rodríguez02] orientada específicamente a sistemas de tiempo real. En esta versión, denominada como MAFALDA-RT, se para el reloj del sistema en el mismo momento de comenzar a ejecutar código de inyección de fallos y se reanuda tras acabar con él. De esta manera desde el punto de vista del reloj del sistema el código de inyección se ha ejecutado en tiempo cero. El inconveniente de esta aproximación es que si bien la intrusión virtualmente desaparece, no se puede utilizar el entorno real del sistema en la validación sino que hay que simularle uno que esté sincronizado con la evolución de la misma.

Como trabajo relacionado también se pueden incluir FIMBUL [Folkesson98] y GOOFI [Aidernark01] ya que, aunque no se trate de inyección software propiamente dicha sí que se utilizan recursos internos al chip. Al tratarse de una inyección basada en Boundary Scan, en realidad el objetivo de esta herramienta son circuitos VLSI, aunque como se describe en [Santos03] estas herramientas se han aplicado a procesadores y en ellas se utilizan los mecanismos internos de depuración del procesador para sincronizar la inyección. En estas herramientas, y en cualquiera que utilice el estándar Boundary Scan actual para llevar a cabo la inyección, nos encontramos ante el mismo problema de la intrusión temporal, ya que para cada acceso a los valores de los pines es necesario leer y escribir toda la cadena de registros.

Actualmente el equipo desarrollador de GOOFI también está utilizando la interfaz Nexus™ para realizar la inyección de fallos, aunque su objetivo principal es llevar a cabo un análisis pre-inyección para obtener datos referentes a cobertura en la ejecución del código y así optimizar el rango de direcciones de memoria donde realizar la propia inyección de fallos [Barbosa05]. En este caso lo que hacen es colocar *breakpoints* (o puntos de ruptura) en las instrucciones que van modificar un contenido de memoria o registro, parando justo antes de que se realice una lectura del dato que va a ser modificado, con lo cual parece ser que antes de la inyección hay una parada del sistema. Esta detención de la ejecución del sistema supone una importante intrusión en el mismo y no resulta en absoluto recomendable, sobretodo si lo que se quiere es evaluar sistemas de tiempo real.

Otra herramienta bastante interesante relacionada con la presente tesis es BALLISTA, cuyo objetivo es realizar tests de robustez de sistemas operativos de propósito general que cumplen con la especificación POSIX [Kropp98]. Para ello se seleccionan un conjunto de llamadas al sistema y se estudia cuál es el tipo de datos de cada uno de los parámetros de éstas. A partir de la experiencia en el campo de la inyección de fallos del equipo de BALLISTA, según se dice en los diferentes artículos al respecto, estos parámetros son sustituidos con unos valores concretos que han demostrado que ciertas llamadas al sistema con valores inválidos en sus parámetros pueden provocar fallos de tipo catastrófico.

Y por supuesto el trabajo más importante relacionado con esta tesis, como es la herramienta INERTE, ya que a partir de la experiencia obtenida en el desarrollo de la misma dentro del GSTF, se ha desarrollado una nueva herramienta que no sólo es capaz de inyectar fallos físicos en zonas aleatorias de memoria en *runtime*, sin el problema de la intrusión temporal, sino que

esta nueva herramienta nos permite realizar tests de robustez de integración de componentes COTS, observando como fallos de diseño de los mismos pueden afectar a los sistemas de tiempo real. En este caso y como veremos posteriormente la inyección de fallos se centra en las APIs utilizadas por los diferentes componentes en su interacción durante la ejecución del sistema. Además, con la nueva herramienta es posible ver el efecto de los fallos en los plazos de las tareas que se ejecutan en el sistema, los tiempos de detección de fallo del propio sistema y otros parámetros de tipo temporal que son relevantes en el estudio de la confiabilidad de un sistema de tiempo real.

3.7 RESUMEN Y CONCLUSIONES

En este capítulo se ha realizado una clasificación de las técnicas de inyección de fallos utilizadas hasta la fecha. Se ha visto que se pueden clasificar como técnicas de inyección de fallos sobre modelos, también llamadas de simulación, y técnicas de inyección de fallos sobre prototipos, que pueden ser basadas en hardware o en software.

Se han descrito cada una de estas técnicas, mostrando las variantes que han aparecido junto con los trabajos más relevantes y se han analizado las ventajas e inconvenientes de cada una de ellas.

Se ha hecho especial énfasis en las técnicas de inyección de fallos software, más cercanas al objetivo de este trabajo. Se ha visto que este tipo de técnicas, basadas en modificar la imagen en memoria de la aplicación, pueden realizar una modificación previa a la ejecución de la misma o una modificación en tiempo de ejecución, dando lugar a diferentes activaciones e interferencias con el sistema sobre el que se aplican.

Por último se han comentado los trabajos más cercanos a los objetivos de la presente tesis. Por un lado está el trabajo existente orientado específicamente a inyección de fallos sobre microcontroladores. Por otro lado, los trabajos que se han llevado a cabo utilizando los mecanismos de depuración incluidos en los procesadores y por último para cada uno de ellos se ha comentado cómo resuelven el problema de la intrusión temporal sobre el sistema a validar.

Tomando estos trabajos como punto de partida, en esta tesis se propone una nueva metodología de inyección de fallos en tiempo real que permita evaluar componentes software COTS que pueden ser utilizados en la implementación de un sistema de tiempo real funcionando sobre un sistema empotrado. Esta técnica se describirá posteriormente en el capítulo 5.

Capítulo 4

INYECCIÓN DE FALLOS CON NEXUS™

4.1 INTRODUCCIÓN

En el capítulo anterior se ha mostrado cuál es el estado del arte de las diferentes técnicas de inyección de fallos y qué herramientas de las actuales se aproximan más a la filosofía que se ha seguido en esta tesis doctoral. A continuación, se va a exponer cómo a través de Nexus™, interfaz de depuración estándar orientada a la depuración de procesadores empotrados [Nexus99], se va a llevar a cabo la inyección de fallos sobre un sistema de tiempo real para la evaluación de su confiabilidad. Más allá del objetivo original de Nexus™ como estándar de depuración, en la presente tesis se propone la utilización de esta interfaz para validar, mediante inyección de fallos, la confiabilidad de una aplicación empotrada construida a partir de la integración de diferentes componentes COTS.

Es un hecho constatado que conforme avanza la tecnología de semiconductores aumenta cada vez más la integración de dichos dispositivos. Se construyen microcontroladores que integran un gran número de periféricos y memorias de diferentes tecnologías y gran capacidad, de forma que el funcionamiento del sistema es cada vez menos visible desde el exterior. Esto hace más difícil diseñar y fabricar herramientas de depuración capaces de implementar las características de controlabilidad y observabilidad necesarias. Una solución que se ha ido adoptando por diferentes fabricantes ha sido la de incluir dentro del procesador los mecanismos necesarios para conseguir estas características. Sin embargo, al no existir un estándar para acceder a estos mecanismos cada fabricante de procesadores ha ido diseñando sus propias interfaces. El estándar Nexus™ es uno de los últimos intentos de conseguir una uniformidad en las herramientas de desarrollo para sistemas empotrados. En la definición de Nexus™ se ha partido de las necesidades que aparecen tanto en la fase de depuración como en la de calibrado de sistemas basados en los microcontroladores actuales [Miller00].

Nexus™ es un estándar definido por el consorcio Nexus 5001 Forum™ compuesto por importantes fabricantes de procesadores, como Hitachi, Infineon, Motorola, National Semiconductor y ST. En este sentido tanto Motorola como Infineon y ST ya tienen disponibles en el mercado microcontroladores con puertos Nexus™ de diferentes clases de conformidad para los que ya existen herramientas comerciales de desarrollo, como las de [Ashling03] [Lauterbach02] [Hitex03].

En los primeros apartados de este capítulo se explica qué es Nexus™ y cuáles son los criterios que se han seguido en su definición. A continuación se enumeran las funciones que se deben implementar en un sistema compatible con Nexus™ y las clasificaciones de las posibles herramientas que se desarrollen según clases de conformidad con el estándar. Por último, se explica cómo se realizaría la inyección de fallos con Nexus™ y los atributos de la misma, así como finalmente las medidas que se obtendría de los experimentos a partir de la información que una herramienta de depuración ofrece.

4.2 SISTEMAS DE DESARROLLO DE APLICACIONES EMPOTRADAS

En primer lugar, haciendo un poco de historia, cabe recordar la evolución que han tenido los sistemas de desarrollo para aplicaciones empotradas en los últimos tiempos. En un principio se popularizó la utilización de programas que monitorizaban al sistema. Esta solución era económica y se sigue utilizando bastante aunque tiene una serie de limitaciones, como son la necesidad de utilizar parte de las memorias ROM y RAM del sistema además de ocupar un canal de comunicaciones, que normalmente suele ser un puerto serie.

El siguiente paso fueron los emuladores clásicos, también llamados ICE (del inglés "*In-Circuit Emulator"*). Estos emuladores a menudo estaban basados en versiones especiales del procesador al que emulan, y disponen de un módulo que se pincha en el sistema sustituyendo al mismo. Algunas características interesantes de los emuladores son la posibilidad de poner puntos de ruptura en la memoria del sistema, facilidades para hacer trazado de la ejecución de la aplicación y el hecho de no utilizar ningún recurso del propio sistema. A cambio en contrapartida suelen tener un elevado precio.

Los emuladores han ido teniendo cada vez más problemas debido a tres causas principales:

- Las cada vez más altas frecuencias de reloj de los procesadores complican el diseño del emulador [Berger03].

- La integración de las memorias en el mismo encapsulado del procesador dificultan al emulador la determinación de la instrucción en ejecución, al no dar una visibilidad externa de la memoria [O'Keefe00].

- Los encapsulados de montaje superficial dificultan el proceso de pinchar un emulador en el zócalo del procesador o en paralelo con él.

Esto ha motivado un desplazamiento progresivo de la circuitería necesaria para hacer una depuración del sistema hacia el interior del procesador, de tal manera que el emulador se quede integrado con el mismo y sólo haga falta un puerto que comunique esta circuitería con el exterior. De esta manera fueron apareciendo soluciones para los diferentes diseños de procesadores, a menudo basadas en el puerto JTAG ya existente en el chip, como los de Cygnal [Cygnal03] o los DSP de Texas Instruments [Texas94].

Algunos fabricantes de semiconductores han intentado seguir una uniformidad en sus diseños para conseguir hacer compatibles entre sí sus sistemas de desarrollo, un ejemplo es el puerto BDM (del inglés "*Background Debug Mode"*) de los procesadores de Motorola [Motorola00].

El trabajo presentado en [Stence03] indica que el 50% del esfuerzo dedicado a desarrollo de aplicaciones empotradas se centra en las herramientas de desarrollo. Por ello el siguiente paso está en conseguir que los procesadores de los diferentes fabricantes comuniquen la información de depuración siguiendo una interfaz estandarizada de tal manera que, con una sola herramienta de desarrollo, se pueda trabajar con cualquier procesador. Según este mismo trabajo, el objetivo de Nexus™ es:

> *"Publicar estándares globales y abiertos de interfaces hardware más software para depuración y calibración de sistemas empotrados; para satisfacer las necesidades que tienen los ingenieros de desarrollo de un soporte reutilizable y potente".*

En la figura 4.1, se puede ver la imagen publicitaria de uno de los sistemas existentes basados en Nexus™ y utilizado en esta tesis doctoral. En esta imagen se puede apreciar cómo se realizan las conexiones entre la herramienta de desarrollo y el sistema empotrado.

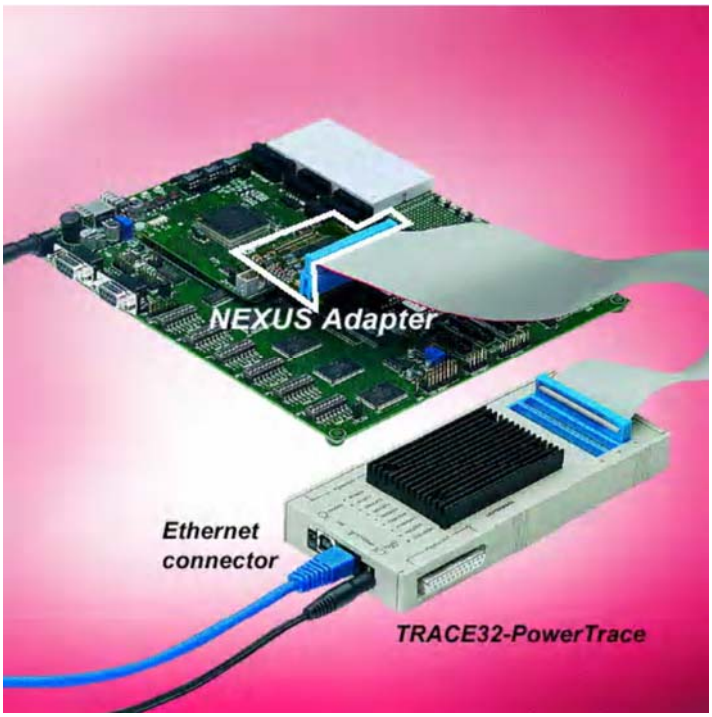


Figura 4.1: Conexión entre herramienta Nexus™ y sistema empotrado.

4.3 HISTORIA DE NEXUS 5001™

En Abril de 1998 se formó el consorcio GEPDIS (del inglés "*Global Embedded Processor Debug Interface Standard*") para definir una interfaz de depuración para aplicaciones de control empotradas. Inicialmente se pensó en un estándar que pudiera satisfacer las necesidades de la industria de automoción, especialmente de las aplicaciones de control de motor. Sin embargo se ha diseñado un estándar de propósito general para desarrollo de software y depuración sobre procesadores empotrados, pensado también para otros tipos de aplicaciones, como son las de telecomunicaciones, periféricos, sistemas inalámbricos, etc. El 23 de Septiembre de 1999 el consorcio GEPDIS se instituyó dentro del IEEE Industry Standards and Technology Organization (IEEE-ISTO) cambiando su nombre al de Nexus 5001 Forum™ publicando la primera versión del estándar IEEE-ISTO 5001™-1999 también conocido como Nexus™. Se

anunció en [Embedded03] la salida de una actualización, corrigiendo inexactitudes de la primera versión y estandarizando detalles como el método de transferencia de información entre el procesador y las herramientas de desarrollo o las funciones comunes de interfaz que hasta entonces no estaban especificadas.

El estándar Nexus™ parte de una interfaz de conexión ya utilizada comúnmente por los fabricantes de microprocesadores, el JTAG o IEEE 1149.1. Se define un puerto auxiliar ampliable que se puede utilizar tanto conjuntamente con el IEEE 1149.1 como de forma independiente. Se definen funciones de los pines del puerto auxiliar, protocolos de transferencia y funciones normalizadas para desarrollo.

4.4 NECESIDADES DE LAS APLICACIONES A LAS QUE VA DIRIGIDO

En Nexus™ se parte de las funciones que se considera deben proporcionar las herramientas de depuración. Primero se tienen en cuenta las funciones básicas de desarrollo orientadas al análisis lógico y al control de ejecución.

Desde el punto de vista del análisis lógico se necesita:

- Acceder a la información de traza con un impacto aceptable en el sistema bajo desarrollo. El ingeniero de desarrollo debe poder relacionar flujo de ejecución de instrucciones con interacciones en el mundo real.
- Recopilar información de cómo fluye la información a través del sistema con un impacto aceptable en el sistema bajo desarrollo, sabiendo qué recursos del sistema están creando y accediendo a la información.
- Validar si el software empotrado se está ejecutando con las debidas prestaciones con un impacto aceptable en el sistema bajo desarrollo.

Desde el punto de vista del control de ejecución se necesita:

- Adquirir y modificar, mientras el procesador está parado, todo el mapa de memoria del procesador en modo supervisor.
- Soportar las características de ***breakpoints*** y ***watchpoints***[1] que tienen los depuradores, bien como *breakpoints* hardware o software, dependiendo de la arquitectura. La configuración de los *breakpoints* y *watchpoints* se debe poder hacer mientras el procesador está parado.

Para que las herramientas puedan proporcionar todas estas funciones con los microprocesadores y microcontroladores actuales los ingenieros de desarrollo de las mismas se encontraron con una serie de problemas. Estos problemas son debidos a la baja visibilidad disponible por la integración en el mismo chip de cachés y memorias FLASH y RAM. El estándar Nexus™ se plantea la necesidad de trabajar específicamente en los siguientes puntos:

- Proporcionar a las herramientas de desarrollo una traza de ejecución de programa con un impacto aceptable en el sistema bajo desarrollo. Con memorias FLASH y cachés de altas prestaciones integradas en el mismo chip se restringe la visibilidad necesaria para las trazas de programa. En algunas aplicaciones no hay bus externo o se utiliza para sus funciones secundarias, como por ejemplo entrada/salida de propósito general.

[1] *"Breakpoint"* o punto de ruptura y *"watchpoint"* o punto de observación.

- Proporcionar a las herramientas de desarrollo una traza de datos con un impacto aceptable en el sistema bajo desarrollo. Con memorias FLASH y cachés de altas prestaciones integradas en el mismo chip se restringe la visibilidad necesaria para obtener trazas de datos. Se consideran dos tipos de visibilidad de datos:

    - Qué proceso (o qué dirección de programa) ha escrito qué parámetro y cuál es el dato que se ha escrito.
    - Para un parámetro concreto, qué procesos han accedido a él.

- Crear una metodología de desarrollo y un conjunto de herramientas estandarizados para desarrollar aplicaciones empotradas. Como los fabricantes de procesadores empotrados normalmente no soportan ni las mismas metodologías ni las mismas interfaces de desarrollo, las herramientas no son compatibles.
- Crear una interfaz de desarrollo estandarizada para poder hacer desarrollos con diferentes núcleos o periféricos en el mismo microcontrolador. La interfaz de desarrollo debería poder controlar cada procesador de forma independiente.
- Crear una interfaz de desarrollo estandarizada independiente para poder trabajar con cualquier arquitectura de procesador.
- Crear una interfaz de desarrollo estandarizada que permita conectar varias herramientas de desarrollo. Podrá ser necesario realizar un arbitraje entre herramientas si varias están conectadas al mismo sistema. En este estándar no se considera el arbitraje entre herramientas.
- El multiplexado de las funciones de desarrollo se debe hacer de forma que no cree complicaciones al ingeniero de desarrollo del sistema. Los fabricantes de microcontroladores a veces multiplexan funciones de desarrollo y entrada/salida de propósito general en los mismos pines. Se deben crear indicaciones para no crear este tipo de conflictos, especialmente al salir del estado de reset, que podrían producir comportamientos impredecibles y anomalías en las herramientas de desarrollo. Los pines dedicados a la interfaz de desarrollo se deberían configurar para estar en modo de desarrollo al salir de reset.
- Se necesita que la interfaz de desarrollo estándar sea escalable, de forma que pueda funcionar para familias de microcontroladores y microprocesadores en diferentes rangos de precio.
- Se necesita una interfaz de desarrollo estandarizada para crear herramientas competitivas.

4.5 FUNCIONES DE NEXUS™

En Nexus™ se definen una serie de características que el sistema debe cumplir. Son las siguientes:

4.5.1 Application Programming Interface (API)

Se define la semántica de los registros recomendados por Nexus™ (NRR)[2], de forma que las herramientas compatibles puedan efectuar un conjunto de operaciones comunes en cualquier sistema, independientemente de su categoría de conformidad o del conjunto de registros que implemente.

[2] Del Inglés "*Nexus Recommended Register*".

4.5.2 Control de Desarrollo y Estado

Se deben implementar los registros recomendados (NRR) u otros que implementen las funciones de desarrollo especificadas en el estándar. Los registros NRR para control de desarrollo y estado son:

- Registro de identificación de dispositivo.
- Registro de selección de cliente.
- Registro de control de desarrollo.
- Registro de estado de desarrollo.
- Registro de dirección base de usuario. Dirección base del mapa de memoria para características de acceso de usuario o de desarrollo.
- Registros de acceso de lectura/escritura.
- Registro de disparo de *watchpoint*.
- Registros de atributos de traza de datos.
- Registros de control de *watchpoints/breakpoints*.

4.5.3 Acceso de lectura/escritura

Este servicio proporciona acceso a todos los dispositivos localizados en el mapa de memoria de usuario tanto cuando el cliente está parado como en tiempo de ejecución. Se puede elegir cualquiera de las dos opciones al implementar esta característica. Se pueden implementar los NRR, lo cual proporcionaría unos registros específicos para dar soporte a este servicio, o se pueden utilizar los mensajes públicos definidos en el estándar.

4.5.4 Traza de identificación de proceso (Ownership Trace Messaging)

La traza de identificación de proceso funciona basándose en mensajes Ownership Trace Messaging (OTM). Se envía un mensaje a la herramienta de depuración cada vez que se activa un proceso o tarea del sistema operativo para identificarla. De esta manera la herramienta de depuración puede conocer el flujo de ejecución de las tareas. Además, para procesadores con memoria virtual, se envían mensajes OTM con cierta periodicidad, a una frecuencia mínima de uno por cada 256 mensajes de traza de programa o de datos.

En el estándar se define un registro OTM. El software de usuario se debe encargar de escribir en este registro para transmitir la información de identificación de proceso. Cuando el sistema operativo escribe en el registro OTM, la información se transmite automáticamente a través del puerto auxiliar.

4.5.5 Traza de programa

El funcionamiento de la traza de programa está basado en un modelo de cambio en el flujo de programa. Cada vez que aparece una discontinuidad en el flujo de programa se envía un mensaje. Estas discontinuidades son las bifurcaciones que se toman y las excepciones. La herramienta de depuración debe interpolar el resto de instrucciones ejecutadas a partir del código objeto estático. Este modelo no es válido para código automodificable, ya que en ese caso el código no es estático.

La información que se envía incluye el número de instrucciones que se han ejecutado desde el último mensaje de bifurcación. La herramienta de depuración utiliza esta información para saber qué salto se ha tomado o qué instrucción ha sido la que ha causado la excepción. Además también se envía un mensaje de traza de programa cada vez que se activa un *watchpoint*.

4.5.6 Traza de datos

La traza de datos define un mínimo de dos ventanas de memoria sobre las que comprobar los accesos de datos. Se definen según dos parámetros:

- Principio y fin de la ventana.
- Envío de mensajes para cada lectura, escritura o las dos dentro de la ventana especificada.

Se comprueban todos los accesos, y para cada uno de los que cumplan las condiciones especificadas se envía un mensaje a través del puerto auxiliar. Se envían la dirección y el contenido accedidos. La dirección se envía de manera relativa al último mensaje de traza de datos enviado.

4.5.7 Sustitución de memoria

Un procesador que proporcione el servicio de sustitución de memoria debe ser capaz de efectuar las siguientes tres operaciones:

- Leer datos e instrucciones desde el puerto auxiliar.
- Leer datos solamente del puerto auxiliar.
- Leer instrucciones solamente del puerto auxiliar.

No es necesario que se puedan hacer escrituras en el puerto auxiliar. Durante las operaciones de sustitución de memoria el procesador accederá a la información en su modo de acceso normal, de la misma manera que accedería a la memoria fuera de estas operaciones. El procesador debe ser capaz de activar la función de sustitución de memoria desde el *reset* o mediante la activación de un *watchpoint*.

4.5.8 Breakpoints/Watchpoints

Los procesadores que proporcionen estos servicios deben ser capaces de generar un mínimo de dos *breakpoints* de instrucciones o datos mediante hardware y de transmitir la ocurrencia de un *watchpoint* a través del puerto auxiliar.

Los *breakpoints* y *watchpoints* incluyen los siguientes:

- *Breakpoints de datos*. El procesador se para en una instrucción cuando se activa un disparo o trigger. Se activa cuando el contenido y/o dirección de un dato se corresponden con los especificados previamente.
- *Breakpoints de instrucciones.* El procesador se para cuando se han ejecutado completamente todas las instrucciones previas y antes de efectuar ningún cambio de estado relativo a la instrucción asociada a la dirección especificada previamente.

- *Watchpoints.* Es un *breakpoint* de datos o instrucciones que no provoca la parada del procesador. Para señalar la condición se envía un mensaje de tipo *watchpoint* a través del puerto auxiliar.

4.5.9 Reemplazo y compartición de puertos

Estas funciones facilitan la utilización de patillas o pines del procesador para la conexión con la herramienta de depuración. El reemplazo de puertos define un mecanismo para sustituir funciones de los puertos del procesador por mensajes enviados a través del puerto auxiliar. Los mensajes normales entre la herramienta de depuración y el procesador proporcionan toda la información necesaria, aunque con un retardo adicional. De esta manera los pines que se conecten a la herramienta de depuración se pueden seguir utilizando para otro cometido.

La compartición de puertos permite que se utilice para el puerto auxiliar un pin dedicado a una función principal del sistema (como un bus externo del procesador). Durante el funcionamiento normal el pin genera las señales de la función principal. Cuando aparece una condición que genera una información, se transmite un mensaje a través de los pines compartidos del puerto y la herramienta de depuración captura dicho mensaje.

4.5.10 Adquisición de datos

Esta función facilita la visibilidad de variables intermedias de la aplicación del procesador. Se envían mensajes a través del puerto auxiliar con la información interna expresada en forma diferencial respecto al envío anterior. De esta forma el envío es más eficiente que mediante trazas de datos.

4.6 CLASES DE CONFORMIDAD Y PRESTACIONES

La capacidad de los puertos de desarrollo conformes con Nexus™ se define desde dos puntos de vista: Las funciones de desarrollo que implementan y las prestaciones en transmisión de información a través del puerto. Existen 4 clases de conformidad y para cada una de ellas se define un conjunto de funciones que se deben implementar. Las prestaciones de transferencia se definen por la posibilidad de transmitir en modos *full-duplex o half-duplex* y por el ancho de banda para los dos sentidos de la comunicación.

Un puerto Nexus™ implementado en un procesador se debe definir de la siguiente forma:

- Conforme a la clase 1, 2, 3 o 4.
- Tasa de transferencia de recepción del procesador (Download rate).
- Tasa de transferencia de envío del procesador (Upload rate).
- *Full* o *half-duplex*.

Desde el punto de vista de las clases de conformidad los procesadores se pueden clasificar como conforme a las clases 1, 2, 3, 4 o alguna subclase aprobada. La clase 1 es la que implementa menos funciones y la clase 4 la más completa. Las clases 1, 2 y 3 se pueden considerar como un subconjunto graduado de las funciones de Nexus™ que puede resultar apropiado para algunas aplicaciones. En las tablas 4.1 y 4.2 se pueden ver qué funciones se deben implementar como mínimo en cada clase.

Desde el punto de vista de las prestaciones de transferencia, como se ha dicho, se debe definir un puerto como "*full* o *half-duplex*", en la tabla 4.3 se puede ver la relación entre clases de conformidad y capacidades del puerto Nexus™.

Tabla 4.1. Clasificación de funciones de desarrollo estáticas según clases de conformidad

Función de desarrollo	Clase 1	Clase 2	Clase 3	Clase 4	Función Nexus™
Leer/escribir memoria en modo depuración	A	A	A	A	Acceso de lectura/escritura
Entrar en modo depuración de reset	A	A	A	A	Control de desarrollo y estado
Entrar en modo depuración desde modo usuario	A	A	A	A	
Salir de modo depuración a modo usuario	A	A	A	A	
Ejecutar una sola instrucción en modo usuario y volver a modo depuración	A	A	A	A	
Parar la ejecución del programa en un *breakpoint* y entrar en modo depuración	A	A	A	A	*Breakpoints/ Watchpoints*

Nota: "A" es una función que se debe implementar vía API.

Tabla 4.2. Clasificación de funciones de desarrollo dinámicas según clases de conformidad

Función de desarrollo	Clase 1	Clase 2	Clase 3	Clase 4	Función Nexus™
Posibilidad de definir *breakpoints* o *watchpoints*	A	A	A	A	*Breakpoints/ Watchpoints*
Identificación de dispositivo	A	A y P	A y P	A y P	Mensaje de identificación de dispositivo
Posibilidad de señalizar la ocurrencia de un evento cuando se activa un *watchpoint*	P	P	P	P	Mensaje de *watchpoint*
Supervisar la identificación de proceso mientras el procesador ejecuta código en tiempo real	-	P	P	P	Traza de identificación de proceso
Supervisar el flujo de ejecución del programa mientras el procesador ejecuta código en tiempo real	-	P	P	P	Traza de programa
Supervisar escrituras en memoria de datos mientras el procesador ejecuta código en tiempo real	-	-	P	P	Traza de datos (sólo escrituras)
Leer/escribir posiciones de memoria mientras el procesador ejecuta código en tiempo real	-	-	A y P	A y P	Acceso de lectura/escritura
Ejecutar programa (código y datos) desde puerto Nexus™ para reset o excepciones	-	-	-	P	Sustitución de memoria

Función de desarrollo	Clase 1	Clase 2	Clase 3	Clase 4	Función Nexus™
Posibilidad de iniciar trazas de identificación de proceso, programa o datos con activación de *watchpoint*	-	-	-	A	Control de desarrollo y estado
Posibilidad de iniciar sustitución en memoria con activación de *watchpoint* o acceso del programa a dirección específica del dispositivo	-	-	-	O	Control de desarrollo y estado
Supervisar lecturas de memoria de datos mientras el procesador ejecuta código en tiempo real	-	-	O	O	Traza de datos (sólo lecturas y escrituras)
Sustitución de puertos de baja velocidad y compartición de puertos de alta	-	O	O	O	Reemplazo y compartición de puertos
Transmitir datos hacia la herramienta	-	-	O	O	Adquisición de datos

Nota:
"A" es una función que se debe implementar vía API.
"P" es una función que se le implementar vía puerto auxiliar mediante mensajes públicos o con un pin del puerto Nexus™.
"O" es una función opcional.

Tabla 4.3. Prestaciones de la interfaz Nexus™

Función de desarrollo	Clase 1	Clase 2	Clase 3	Clase 4	Función Nexus™
Puerto IEEE 1149.1	X	-	-	-	Half duplex
Puerto IEEE 1149.1 o puerto auxiliar de entrada con puerto auxiliar de salida	-	X	X	X	Full duplex

4.7 INYECCIÓN DE FALLOS CON NEXUS™

Una vez se ha visto qué es Nexus™ y qué capacidades y características tiene como estándar orientado a la depuración de procesadores empotrados [Nexus99], en la presente tesis se propone la utilización de dicha interfaz para validar, mediante inyección de fallos, la confiabilidad de una aplicación empotrada.

Lo que se pretende es conseguir una herramienta que implemente la metodología que se propone, y que se explicará en el siguiente capítulo, que sea externa al sistema a validar al tiempo que se pueda conectar al mismo para obtener información relativa a su confiabilidad, con el objetivo de no introducir ninguna sobrecarga temporal, para así respetar las restricciones de tiempo real impuestas. Además ésta debe ser fácilmente utilizable para evaluar diferentes componentes COTS que pudieran ser integrados en diferentes sistemas.

Esta herramienta consistiría en un computador personal, una interfaz Nexus™ que permita a este computador comunicarse con la circuitería Nexus™T del sistema empotrado y el software encargado de dirigir los diferentes experimentos y analizar los resultados. En la siguiente figura 4.2 se puede ver el esquema general de cómo sería la organización de dicha herramienta. A continuación se verán en primer lugar las características de Nexus™, de las ya comentadas, que se pueden utilizar para implementar una técnica general de inyección de fallos que evite cualquier interferencia o intrusión temporal sobre el sistema a validar, a la vez que permita

observar su actividad. Posteriormente se describe la metodología a seguir para realizar una inyección de fallos basada en Nexus™.

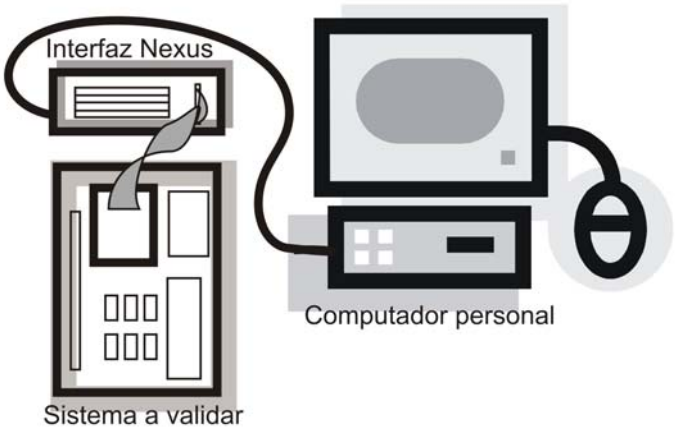


Figura 4.2: Inyección de fallos con Nexus

4.7.1 Portabilidad

Uno de los objetivos del consorcio Nexus™ es la especificación de una interfaz universal de depuración. Esta interfaz posibilita el desarrollo de herramientas de depuración capaces de funcionar sobre cualquier sistema que incorpore un puerto compatible con Nexus™. Como la técnica de inyección de fallos que se busca debe ser portable y dado que la especificación Nexus™ es un estándar, las herramientas que se desarrollen siguiendo esta técnica se beneficiarán de esta portabilidad. De esta manera, estas herramientas serán aplicables a cualquier sistema empotrado compatible que incorpore la interfaz Nexus™.

4.7.2 Sobrecarga temporal

Como se ha dicho en el capítulo 3, las herramientas de inyección de fallos software introducen en el sistema una sobrecarga que puede alterar (entre otras cosas) el comportamiento temporal del sistema a validar. Además como se ha visto en otras herramientas, muchas veces se recurre a parar la ejecución del sistema para realizar la inyección. Para resolver estos problemas, se puede hacer uso de ciertos servicios proporcionados por la interfaz Nexus™ como son:

- El acceso a memoria en tiempo de ejecución, que se puede utilizar para corromper el contenido de la memoria sobre la marcha (del inglés "*on-the-fly*"), es decir sin parar la ejecución. Este tipo de acceso es completamente transparente desde el punto de vista del reloj del sistema empotrado y por tanto no provoca ninguna sobrecarga temporal sobre el mismo.

- Y los *watchpoints*, que se pueden utilizar para disparar la inyección de fallos. Los *watchpoints* son eventos de señalización de la circuitería de depuración de Nexus™ que no paran la ejecución de la aplicación para indicar algún tipo de evento, como pueda ser la efectividad de la inyección o una marca temporal para calcular algún tiempo de latencia. La inyección de fallos se puede disparar en función de una serie de condiciones basadas en modos de acceso a la memoria. Por ejemplo, cuando se ejecute el contenido de una dirección de memoria predeterminada o cuando la aplicación

acceda en lectura o escritura a la dirección de memoria que contenga el dato que se quiere supervisar.

4.7.3 Observabilidad

Como el objetivo principal de la especificación de Nexus™ es la depuración, éste ofrece una serie de servicios muy útiles y precisos para la observación del sistema. De cara a observar los efectos producidos por la inyección de fallos sobre el sistema se pueden utilizar las siguientes funciones de Nexus™:

- La traza de programa y los *watchpoints* se pueden utilizar para saber si se ha activado o no un error en memoria, como se ha visto anteriormente. Los mensajes de la traza de programa se envían cuando hay discontinuidades en el flujo de ejecución del programa, como saltos o excepciones. La herramienta de depuración puede reconstruir una traza completa de la ejecución de la aplicación a partir de estos mensajes. Si el flujo de ejecución de la aplicación pasa por encima de la posición de memoria modificada, se ejecutará la instrucción que contenga. Esto provocará la activación del error. En este caso, la traza de programa detectará esta condición y almacenará toda la información relativa a este evento.

- La traza se puede seguir para conocer cuál es el comportamiento del sistema tras el error, si el error se ha detectado o no, e incluso si el sistema tolera el error se puede calcular el tiempo de recuperación y tiempos de latencia. La traza de datos se puede utilizar para saber si una inyección en memoria de datos se ha activado (con una traza de lectura) o si la aplicación ha vuelto a escribir un dato en la posición de memoria modificada antes de activarse el error (con una traza de escritura).

- Los *watchpoints* también se pueden utilizar para marcar eventos especiales del sistema en la traza para su análisis posterior. Un ejemplo podría ser el marcar el instante en que el error se hace efectivo y el instante de recuperación para medir el tiempo entre ambos.

- En última instancia los *watchpoints* también pueden ser utilizados para monitorizar el contenido de ciertas variables del sistema, como aquellas que notifican códigos de error, o variables que nos indican el comportamiento de salida del sistema. Además el tiempo en el que éstas son modificadas también puede ser obtenido con la finalidad de calcular, en el caso de variables de error los tiempos de detección de errores, o en el caso de variables de salida del sistema los tiempos de repuesta.

4.8 ATRIBUTOS DE LA INYECCIÓN DE FALLOS

A continuación una vez vistas qué características de Nexus™ que se pueden utilizar en favor de una inyección de fallos no intrusiva, en este punto se van a exponer los atributos de la metodología de inyección de fallos que posteriormente se explica y que permite inyectar, tanto fallos de tipo hardware, inyectando en zonas aleatorias de memoria, como fallos de tipo lógico para detectar fallos de diseño del software que puedan surgir como consecuencia de la integración de diferentes componentes COTS, que pudieran ser utilizados para la construcción de un sistema empotrado de tiempo real.

A continuación se muestra cómo se realiza la inyección de fallos viendo cuál es el modelo de fallo por el que se ha optado, y cuándo se inyecta el fallo definiendo el disparo de inyección.

Además se mostrará dónde se inyecta el fallo para determinar la localización del mismo y cómo obtener medidas a partir de los experimentos en inyección de fallos.

4.8.1 Modelo de fallo

El modelo de fallo que se ha considerado es el del tipo "inversión" o "*bit-flip"* que, como se ha demostrado en [Gil02], es el más realista para emular fallos en el hardware, pero también para inyectar fallos en el software [DBench04], que son el objetivo del presente trabajo puesto que lo que se pretende es abordar la problemática que surge al integrar diferentes componentes COTS.

Para inyectar fallos en el software, la inyección de fallos software debe realizarse en determinadas zonas de memoria con el objetivo de emular posibles fallos de diseño del software que no han sido detectados en fases previas de testeo del mismo (fallos residuales del software). Estas zonas de memoria a las que se hace mención, básicamente se corresponden con los parámetros de la API (del inglés "*Application Programming Interface*") de los servicios que ofrece un componente COTS cualquiera que quisiéramos incorporar a nuestro proyecto, y así poder comparar sistemas diferentes. Puesto que como recomienda la norma [ISO/IEC 61508], se supone un punto de vista de caja negra para los componentes, por lo que la interfaz de los mismos será el punto de inyección.

Para ello hay dos posibles técnicas de inyección de fallos en dichos parámetros de las llamadas o servicios, como son la sustitución selectiva de valores en los mismos, donde se sustituye el contenido de éstos por otros valores críticos para el sistema; o la técnica del *bit-flip*, cambiando el valor de algún bit de algún parámetro de una llamada al sistema o función de la API del componente. Dentro del Proyecto Europeo DBench [DBench04] estudios han demostrado la equivalencia de ambas técnicas en la obtención de resultados [Madeira03], si bien para la técnica del *bit-flip* hacen falta más experimentos que con la técnica de sustitución selectiva, utilizada ya en otros trabajos como BALLISTA [Kropp98], al final los resultados que se obtienen son equivalentes.

Para inyectar un fallo del tipo *bit-flip* (inversión) lo primero que hay que hacer es leer el contenido de memoria a modificar, aplicarle una máscara XOR y el resultado volverlo a escribir en dicha zona. Por ejemplo, si en una zona de memoria escribimos el resultado de aplicar la función XOR 00000001_b a su contenido estaremos invirtiendo el valor del bit de menos peso. Si la función que aplicamos es XOR 00011100_b estaremos invirtiendo los bits 2 a 4. Variando de esta forma la máscara a aplicar se puede inyectar fallos de tipo inversión simples o múltiples.

Por otro lado, con simplemente cambiar la función que se aplica se pueden inyectar otros tipos de fallos, como el "pegado a" o "*stuck-at"*. El fallo de tipo pegado a 1, se inyectaría aplicando una máscara OR, o el pegado a 0 que se inyectaría aplicando una máscara AND. Por ejemplo OR 00000010_b pone a 1 el bit 1, AND 00011111_b pone a 0 los bits 5, 6 y 7. Este mecanismo se puede ver en la siguiente figura 4.3.

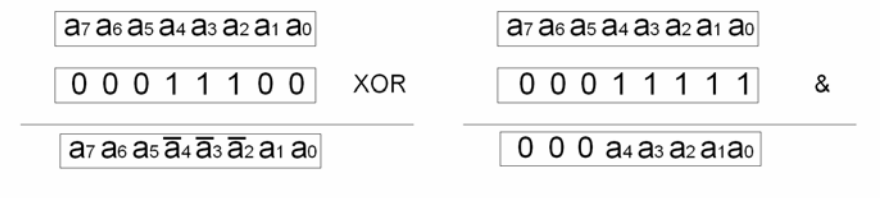


Figura 4.3: Mecanismo para inyectar fallos de tipo inversión y pegado a cero

4.8.2 Disparo

El disparo de la inyección se puede especificar de manera espacial, temporal o una combinación de ambas. Para realizar el disparo de manera temporal sólo se tiene que lanzar a ejecución la aplicación y hacer que la herramienta de inyección espere el tiempo especificado antes de modificar el contenido de la posición de memoria sobre la que se va a inyectar. Si se quiere disparar la inyección de manera espacial se debe poner un *watchpoint* sobre la dirección en la que se desea que se dispare la inyección. Se puede definir un disparo más genérico como la combinación de un disparo espacial, para sincronizar la inyección con un evento del sistema más un retardo. Así, un disparo puramente temporal se podría definir como un retardo después del evento reset y un disparo puramente espacial se podría definir como un retardo cero tras un evento del sistema.

4.8.3 Localización

En las aplicaciones empotradas es de esperar que se produzcan fallos por la integración de componentes COTS provenientes de diferentes fabricantes, fallos que no se han encontrado en fases previas de pruebas de los mismos; pero además también es de esperar un número considerable de fallos físicos debido a las altas densidades de integración que se van alcanzando día a día [Gil02]. Es por ello, que aunque en principio el objetivo que se plantea se centra en los fallos software no se puede dejar de incorporar en la metodología que se propone la posibilidad de inyectar fallos de tipo físico. Hay que remarcar que la mayoría de las técnicas de inyección de fallos software, tanto la presente como las ya conocidas, permiten inyectar tanto fallos físicos como software [Duraes03].

Dado que el objetivo es inyectar fallos de diseño del software, como se ha comentado, la inyección se llevará a cabo en zonas específicas de memoria que se corresponden con ciertos parámetros de la API de los componentes COTS. Aunque posteriormente se explica detalladamente cómo se realiza la selección de dichas zonas de memoria, en realidad la inyección de fallos se debería llevar a cabo en los registros internos del sistema. En dichos registros internos es donde la mayoría de compiladores realizan la carga de los parámetros de las funciones. Este procedimiento de utilización de los registros internos del microcontrolador, se lleva a cabo con el objetivo de acelerar la ejecución de las aplicaciones que se desarrollan para sistemas empotrados. Esto viene impuesto, en este tipo de sistemas, por la utilización de la interfaz estándar para aplicaciones empotradas conocida como EABI (del inglés ”*Embedded Application Binary Interface*”) [EABI98].

Es decir, que el paso de parámetros de las funciones de los componentes en realidad se lleva a cabo en los registros internos del procesador. Pero el problema es que la especificación Nexus™ impone que para sustituir en un momento determinado el contenido de dichos registros internos, que contienen el valor de los parámetros de las funciones, hay que detener la ejecución del sistema, cosa que no sucede con la sustitución de valores en zonas de memoria. Por tanto, el inyectar fallos en los registros internos del sistema supondría producir la consiguiente intrusión en el funcionamiento en tiempo real del mismo y por tanto un método alternativo no intrusivo tiene que ser propuesto (posteriormente en el capítulo 5 dicho método es explicado).

La gran ventaja de utilizar Nexus™ para inyectar fallos es precisamente la posibilidad de inyectarlos en memoria (o en registros especiales accesibles desde el mapa de memoria) en tiempo de ejecución para la evaluación de aplicaciones de tiempo real. Al tener acceso al mapa de memoria interno en tiempo de ejecución no es necesario parar en ningún momento el funcionamiento de la aplicación. De esta manera se elimina la intrusión temporal que la herramienta ocasionaría. Para inyectar un fallo en memoria se puede poner en marcha la aplicación y, llegado el momento de la inyección, escribir la palabra modificada en la posición

de memoria que debe contener el fallo y todo ello como se ha dicho sin detener la ejecución del sistema.

La limitación de Nexus™, al menos en las implementaciones actuales, está al realizar la inyección en los registros internos del procesador, como se ha comentado anteriormente. Para acceder a los mismos es necesario poner al procesador en estado de parada (del inglés "*Halt*"). Al entrar en este estado se está deteniendo momentáneamente la ejecución y, por tanto, hay que estudiar con detenimiento la posible interferencia sobre el sistema antes de aplicarlo a la inyección sobre sistemas de tiempo real. Este problema se puede resolver de varias formas. En primer lugar, cabe decir que es un problema de las implementaciones actuales de Nexus™ en los procesadores de Motorola, y que el estándar en la versión actual da libertad a los fabricantes para definir el acceso a los registros específicos de cada implementación. Es de esperar, por lo tanto, que de cara a conseguir una uniformidad en las interfaces, en futuras implementaciones de Nexus™ en procesadores este problema desaparezca.

En cualquier caso, una posible solución podría ser la adoptada en [Rodríguez02]. En este trabajo se utiliza una inyección de fallos software para validar un sistema de tiempo real y se utiliza un mecanismo de "congelación del mundo" que evita que se pierda la sincronización del sistema con el entorno. Esta opción podría ser válida en sistemas en los que se utilice un entorno virtual o simulado. Pero ¿qué pasa cuando lo que se quiere es evaluar sistemas reales con fuertes restricciones temporales? Llegados a este punto se debería encontrar un solución para inyectar fallos en cualquier parte del sistema sin producir ningún tipo de perturbación, como proponen estándares como IEC 9126-1 que establecen la idoneidad de evaluación de sistemas reales funcionando en condiciones reales. En definitiva la metodología que se propone salva dicho problema pudiendo inyectar sobre zonas de memoria sincronizando el disparo de manera temporal o espacial sin intrusión alguna.

4.8.3.1 Inyección sobre memoria, sincronización temporal

En el diagrama de flujo de la figura 4.4 se puede apreciar cómo se puede realizar una inyección sobre memoria sincronizándola temporalmente, en caso de que la inyección de fallos sea *runtime* (en tiempo de ejecución) [Yuste03e]. Primero se debe aplicar un reset al sistema para asegurar que el estado inicial es conocido, se carga la aplicación, se establecen los mecanismos de observación necesarios y se arranca. Anteriormente a esto y como se explica en el siguiente capítulo se realizaría un análisis previo de la aplicación para obtener las direcciones de memoria donde llevar a cabo la inyección de fallos.

Una vez arrancada la aplicación, en ese momento se inicia un temporizador en el computador encargado de dirigir el experimento. Se espera el tiempo prefijado para el mismo y al vencer dicho temporizador se hace la inyección leyendo la posición de memoria sobre la que se va a inyectar el fallo, modificándola con un fallo de tipo inversión o pegado y volviendo a escribirla en la memoria, todo ello sin necesidad de parar la ejecución como se ha comentado.

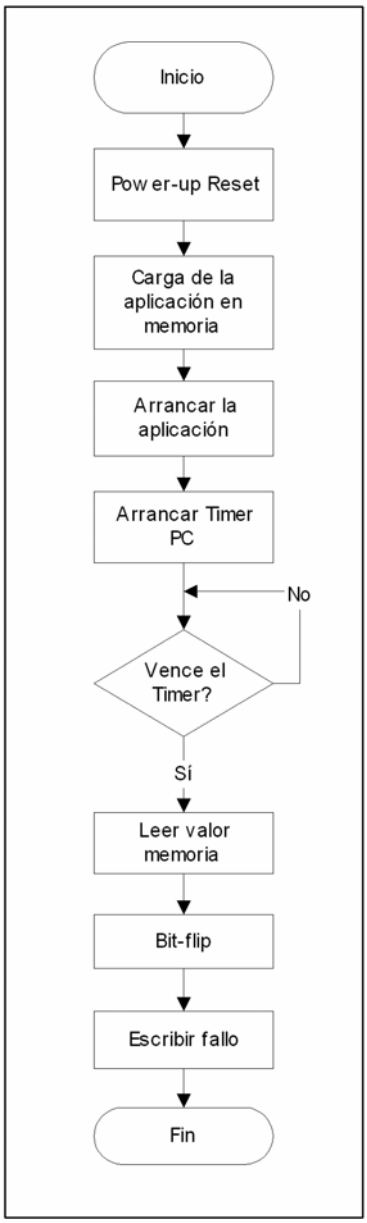


Figura 4.4: Inyección sobre memoria con sincronización temporal

4.8.3.2 Inyección sobre memoria, sincronización espacial

En este caso, previamente a lanzar la aplicación se debe programar la dirección con la que se quiere sincronizar el disparo. Puede ser la misma dirección a inyectar o una dirección que se tome como referencia dentro de la aplicación, por ejemplo el inicio de algún bucle de control.

Los pasos para realizar una inyección sobre memoria utilizando una sincronización espacial se pueden ver en el diagrama de la figura 4.5

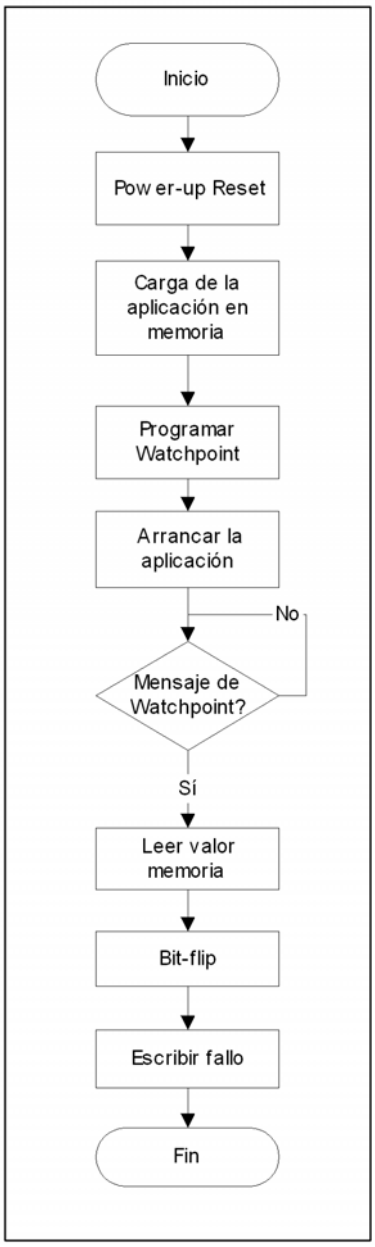


Figura 4.5: Inyección sobre memoria con sincronización espacial

4.8.3.3 Caso general, inyección sobre memoria

Pero si el objetivo es obtener un disparo más genérico como combinación de un disparo espacial más uno temporal, para conseguirlo se puede proceder de la siguiente manera. Al igual que en el caso de sincronización espacial, previamente al inicio de la aplicación hay que programar un *watchpoint* para sincronizar la inyección con un evento del sistema. Al activarse el *watchpoint* se inicia el temporizador y el proceso sigue como en el caso de la sincronización temporal. Es decir, al vencer el temporizador se activaría la inyección. Esto se puede ver el diagrama de flujo correspondiente en la figura 4.6.

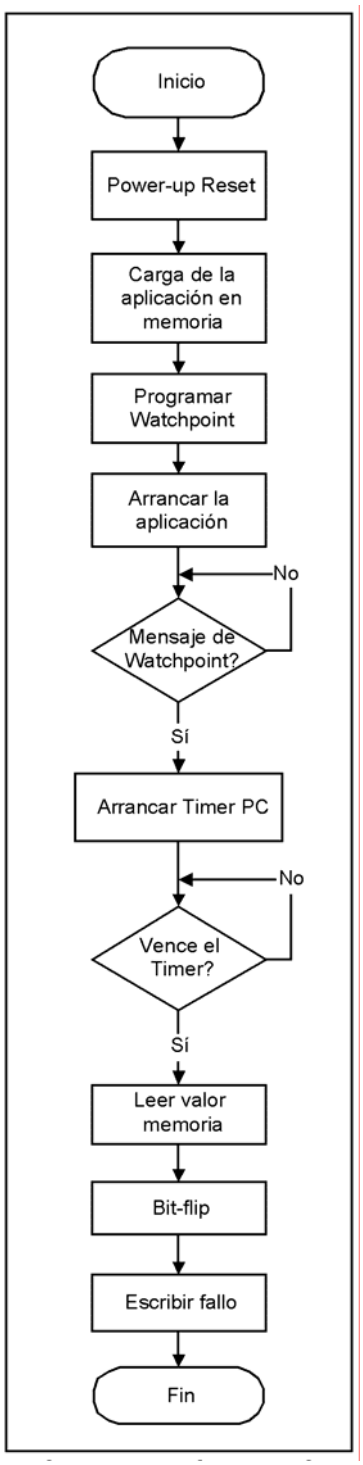


Figura 4.6: Inyección sobre memoria con sincronización combinada

Aunque todo esto siempre y cuando se hable de una inyección *runtime*, es decir cuando el fallo se inyecte en tiempo de ejecución. Pero uno de los problemas de la inyección *runtime* es la baja tasa de efectividad de los fallos introducidos, donde para obtener resultados representativos hace falta llevar a cabo una cantidad considerable de experimentos.

Una alternativa a la inyección *runtime* es la inyección de fallos *pre-runtime* u *off-line*, como se ya se ha explicado en el capítulo 3. En este caso la inyección del fallo se realizaría antes de lanzar a ejecución la aplicación. Está técnica aporta muchas ventajas como una mayor

efectividad de las campañas de inyección, puesto que se puede asegurar que la aplicación ejecutará la dirección de memoria donde se ha introducido el fallo; y una menor intrusión en el sistema bajo prueba, puesto que en este caso no hace falta programar ningún *watchpoint* o temporizador adicional para llevar a cabo la inyección de fallos.

Por tanto, para llevar a cabo una inyección de fallos *pre-runtime* basta con realizar un reset del sistema, establecer todos los mecanismos de observación necesarios, posteriormente realizar la inyección del fallo en la zona de memoria seleccionada y finalmente lanzar a ejecución el sistema para observar su funcionamiento. En este caso, como se ha dicho anteriormente, se asegura la activación del fallo puesto que éste es introducido antes de lanzar a ejecución la aplicación.

4.8.4 Medidas

Respecto a las medidas a obtener, primero se debe establecer una clasificación genérica de los posibles resultados de los experimentos. En la siguiente figura 4.7 [Rodríguez02] se puede ver esta clasificación.

Primero se tiene el conjunto **I** que comprende la totalidad de los experimentos de inyección realizados. Algunos de estos experimentos modifican posiciones de memoria no utilizadas por la aplicación, al no utilizarse estas posiciones, las inyecciones nunca llegan a activarse y por tanto no se consideran como inyecciones efectivas. Las inyecciones no efectivas son las que están en la zona **a**.

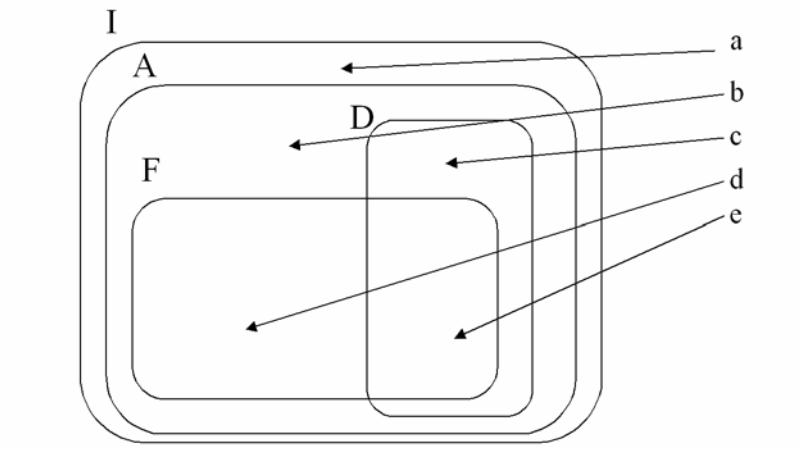


Figura 4.7: Clasificación de los experimentos según resultados obtenidos

El conjunto de inyecciones efectivas, o que se han activado, es el conjunto A. Dentro de él se tiene el conjunto **F** de inyecciones que provocan una avería y el conjunto **D** de inyecciones que activan algún mecanismo de detección de errores.

Si se considera primero una serie de experimentos en los que no se activa ningún mecanismo de detección de errores ni se observa ninguna avería, es decir, que no pertenecen al conjunto **D** ni al **F**. En ese caso se dice que el fallo está cubierto por la redundancia intrínseca del sistema. Estos experimentos son los señalados como **b**.

También hay experimentos que pertenecen al conjunto **D**, que activan mecanismos de detección de errores, pero no provocan ninguna avería. Estos fallos están en la zona marcada como **c** y también se les podría considerar como fallos cubiertos por la redundancia intrínseca del sistema dado que no provocan averías. Como estos fallos disparan los mecanismos de

detección de errores se corre el peligro de sobrestimar la cobertura de los mismos incluyendo fallos que no desencadenan una avería.

Por otro lado se tienen los fallos que pertenecen al conjunto F pero no al **D**, los marcados como **d**. Estos fallos provocan averías no seguras del sistema. Por otro lado, también se tiene los que pertenecen tanto al conjunto **F** como **D**, marcados como **e**, que provocan averías y además activan mecanismos de detección de errores. A las averías en este conjunto las consideraremos como averías seguras.

Sobre la duración de los experimentos se tiene que tener en cuenta varias consideraciones:

- Según [Rodríguez02] si la avería se produce antes de que se active el mecanismo de detección de errores, el mismo no resulta muy útil; por tanto en los experimentos en los que sucede esto se debe considerar que el sistema ha tenido una avería no segura. En este caso se debería parar un experimento en el momento en el que se detecta una avería y considerarlo como del grupo **d** independientemente de que más tarde se active un mecanismo de detección de errores.

- Para evitar sobrestimar la cobertura de detección de errores se debe determinar si un experimento en el que se ha activado un mecanismo de detección de errores está en el conjunto **c** o **e**. Para ello, según [Steininger02] el experimento no debe finalizar con la activación de los MDE (mecanismos de detección de errores), sino que se debe seguir ejecutando la aplicación, ya que los fallos podrían quedar cubiertos por la redundancia intrínseca del sistema.

- La falta de una avería puede ser debida a dos causas: que el error se haya sobrescrito o que el error se quede latente y se manifieste más tarde. En [Chevochot01] se hacen experimentos de más de 20 segundos y se observa que los resultados no varían en cuanto a manifestaciones de averías.

La ejecución de los experimentos hasta después de haberse activado los MDE merece dos puntualizaciones:

- Para el caso de los MDE correspondientes a los componentes software, tras la detección del error la ejecución del sistema debe continuar para ver si los mecanismos de detección y tolerancia a fallos del propio componente u otros componentes a los cuales el error se pudiera haber propagado, son capaces de absorber o tolerar el error introducido y no producir una avería del sistema.

- En el caso de utilizar las excepciones como MDE hay que observar con detalle el comportamiento de las mismas. En algunas arquitecturas, como la arquitectura PowerPC de Motorola [Motorola99], para favorecer la recuperación del sistema, todas las excepciones que sirven de MDE se manejan de forma ordenada. Esto quiere decir que antes de ejecutar el manejador de una excepción todas las instrucciones anteriores a la que provocó esa excepción y los correspondientes manejadores de las posibles excepciones que hubieran provocado se deben haber ejecutado completamente. Además si la ejecución de una instrucción provoca varias excepciones, la arquitectura asegura que los manejadores correspondientes se llamarán secuencialmente. Este comportamiento se debe tener en cuenta a la hora de clasificar los MDE que puedan activarse frente a un error.

4.8.4.1 Obtención de las medidas a partir de la información de la herramienta de depuración

Comparando las trazas de ejecución de las variables de salida en funcionamiento sin inyección y con inyección se puede determinar si se ha manifestado una avería. Si aceptamos como definición de avería la desviación del servicio entregado por el sistema respecto al especificado, tal y como se ha enunciado en el capítulo 2, entonces si las variables de salida son diferentes de las que se obtienen funcionando correctamente se dice que se ha manifestado una avería en el sistema. Al mismo tiempo diremos que desde el punto de vista temporal, si aún a pesar de haberse producido valores correctos en las variables de salida, éstos se han producido en instantes de tiempo que quedan fuera de los rangos permitidos definidos por las especificaciones funcionales del sistema, también diremos que se habrá producido una avería.

Con todo ello habrá que ver si tras la inyección de fallos, durante la experimentación se ha activado algún mecanismo de detección de errores de alguno de los componentes, tanto software como hardware, que integra el sistema. Para detectar las activaciones de los MDE se tienen diferentes alternativas, se puede programar un *watchpoint* en una posición de memoria que contenga una instrucción que se ejecute sólo cuando el MDE detecte el error. Si los MDE que se consideran son los internos del procesador y lanzan excepciones al activarse, se puede programar el *watchpoint* en los puntos de entrada de las excepciones. Si lo que se considera son los MDE de los componentes COTS, en estos casos habrá que observar sus mecanismos de notificación de errores, que normalmente realizan a través de una variable de su interfaz, debiendo poner por tanto también el correspondiente *watchpoint*. De esta forma en las trazas de ejecución se puede ver si se ha accedido en lectura o escritura a las posiciones de memoria marcadas con los correspondientes *watchpoints*, lo que es señal de que se ha activado o no el correspondiente MDE. Además dichos *watchpoints* nos dan la posibilidad de observar el tiempo en que estos se han llevado a cabo y así poder calcular tiempos de detección de errores.

Puesto que el principal objetivo es observar el funcionamiento de los diferentes componentes software frente a fallos de diseño y ver como actúan los MDE de los mismos, determinar si el fallo introducido al final es efectivo es tan simple como monitorizar que la instrucción, que realiza el paso del parámetro en el que se ha inyectado el fallo, se ejecuta. Recordar que la inyección en este caso se realizará sobre los parámetros correspondientes a la API del componente bajo estudio y por tanto se debe observar si el flujo de programa pasa por la posición de memoria que se ha modificado.

Por tanto, para el caso de inyección de fallos tanto de tipo software como de hardware sobre zonas de memoria de código, aunque no se puede realizar una traza de datos para obtener el contenido de la posición de memoria sobre la que se está trazando, ya que el acceso que se realiza no es de lectura sino de ejecución, si que se puede obtener el instante en que se ejecuta una palabra. Así que se tiene la posibilidad de programar un *watchpoint* para que sobre la traza se reflejen los instantes en que se ejecuta la posición de memoria modificada. En caso de que sobre la traza aparezca una activación del *watchpoint*, entonces se puede decir que el fallo se ha activado.

Para el caso de fallos de tipo hardware sobre memoria de datos, determinar si el fallo es efectivo es más complicado. Se puede hacer una traza de datos de forma que se reflejen sobre la traza todos los accesos en lectura de la posición de memoria inyectada. Comparando la traza sin fallo con la traza con fallo se puede determinar si se ha realizado alguna lectura de esa posición en la que el contenido de la memoria sea diferente al contenido que tenía sin inyección de fallos. En ese caso la inyección es efectiva y además en la traza se refleja de manera precisa el instante temporal en que esto se ha producido.

Para el caso de los mecanismos de detección del componente, las latencias se medirían con el cálculo de la diferencia que hay entre el instante en el que se inyecta el fallo hasta que el MDE del componente notifica el error a través de la escritura de la variable correspondiente que implementa ese mecanismo de detección y que posteriormente se explica.

Con todo esto, las medidas que se buscan son latencias, coberturas de detección de errores frente a fallos de diseño del software y tiempos máximos de ejecución para las tareas. Para las coberturas basta con detectar si el fallo inyectado ha producido un error y éste ha evolucionado en avería del sistema. Para las latencias se debe determinar los instantes de activación y detección y calcular la diferencia entre ellos. Para los tiempos máximos de ejecución de las tareas se debe observar si los fallos introducidos al final han afectado a los instantes de tiempo de la entrega del servicio, considerando que el servicio en cuanto a valor es correcto. Por lo tanto se debe determinar, para cada experimento:

- Si se activa el error y el instante de activación.
- Si se produce una avería.
- El instante de detección del error.
- Los tiempos máximos de ejecución de las tareas.

Para obtener estas medidas con la herramienta los resultados que se deben buscar de cada experimento son:

- Para determinar la activación del error:
    - En memoria de código: Traza de ejecución con *watchpoint* sobre el punto donde se realiza la inyección. Si sobre la traza aparece el *watchpoint*, el fallo inyectado es efectivo. La marca temporal del *watchpoint* nos da el instante de activación.
    - En memoria de datos: Traza de lectura de la posición inyectada. Si en la traza aparece un acceso en lectura con valor diferente al de la ejecución sin fallos, el fallo inyectado es efectivo. La marca temporal del *watchpoint* nos da el instante de activación.
- Para determinar si se produce una avería:
    - Traza de escritura en la zona de memoria que contiene la salida del sistema. Si en esta traza aparecen diferencias entre el experimento con inyección de fallos y el experimento libre de fallos, se ha producido una avería en el dominio del valor.
    - Traza de tiempos máximos de ejecución de las tareas: si en la traza se observa que los tiempos de ejecución de las tareas en el peor de los casos (tiempos máximos de ejecución) superan los rangos de tiempos permitidos para éstas, entonces la entrega del servicio desde el punto de vista temporal es errónea y por tanto se ha producido una avería en el dominio del tiempo.
- Para determinar el instante de detección del error:
    - En el componente: Si el error se ha activado, leer directamente de la traza de escritura la marca temporal del acceso a la variable que implementa el MDE. Posteriormente calcular la diferencia entre el tiempo que marca la traza de ejecución con *watchpoint* sobre la dirección donde se ha inyectado el fallo y el

tiempo que marca la traza de escritura con *watchpoint* de los MDE del componente.

- En caso de utilizar los MDE internos del microcontrolador, los *watchpoints* se deberían poner en la entrada de las excepciones. Al igual que antes basta con leer directamente de la traza el tiempo del acceso que ha provocado la activación del *watchpoint*.

- Para determinar los tiempos máximos de ejecución de las tareas:

  - Traza de los tiempos máximos de ejecución de las zonas de código correspondientes a las diferentes tareas que se ejecutan en el sistema. En este caso la traza se obtiene para aquellos experimentos en los cuales los valores de las variables de salida son correctos.

4.9 RESUMEN Y CONCLUSIONES

En este capítulo se ha visto cuáles son las características de Nexus™ que se pueden aprovechar para llevar a cabo la inyección de fallos tanto de software como de hardware. Estas características son el acceso a memoria, tanto de código como de datos en tiempo de ejecución y los mecanismos de *watchpoint* ideales para conseguir la observabilidad necesaria, y así poder eliminar la sobrecarga temporal propia de las herramientas de inyección de fallos. Además el hecho de que Nexus™ es un estándar proporciona portabilidad a las herramientas que utilicen estos mecanismos.

En la segunda parte del capítulo se estudia de forma más genérica los diferentes pasos a seguir para inyectar fallos con Nexus™. Los modelos de fallo pueden ser la inversión o "*bit-flip*" y el pegado o "*stuck-at*" de uno o varios bits. La inyección se puede disparar de diferentes maneras. La primera manera es espacial, es decir, sincronizada con la ejecución de la aplicación. La segunda manera es temporal, esperando un tiempo prefijado desde el inicio de la ejecución. La tercera es una combinación de las dos técnicas anteriores, consistente en esperar un tiempo prefijado desde un evento en la ejecución de la aplicación, todo ello si se opta por una inyección *runtime*. La alternativa es la inyección *pre-runtime* que se ha visto que con las características que nos ofrece Nexus™ se posiciona como una opción muy interesante para una obtención rápida de resultados y con altas tasas de efectividad en los fallos inyectados.

Por último se estudia qué medidas se pueden obtener y cómo obtenerlas a partir de la información que dan las trazas de Nexus™. Para cada experimento se puede obtener la siguiente información: si se ha activado el fallo, si se ha detectado un error, si el error se ha propagado a una avería, el instante de activación del error, el instante de detección del error y los tiempos en el peor de los casos de la ejecución de las tareas del sistema para observar averías que se producen el no cumplimiento de los tiempos de entrega del servicio. Aunque en este capítulo se ha visto de forma general como se puede utilizar Nexus™ para la inyección de fallos, en el siguiente capítulo se explica con mayor detalle cómo utilizar dicho interfaz para inyectar fallos de diseño del software en la interfaz de componentes COTS y cómo calcular la información temporal asociada a la ejecución del sistema bajo la influencia de los fallos inyectados.

Capítulo 5

VALIDACIÓN DE LA ROBUSTEZ DE COTS MEDIANTE LA INYECCIÓN DE FALLOS

5.1 INTRODUCCIÓN

Desde hace tiempo, la reutilización de software ha venido siendo una práctica común para la construcción de productos software. La reducción de los costes, tiempos y esfuerzos en los procesos de elaboración han sido algunos de los motivos que han llevado a los ingenieros de software a considerar técnicas para la reutilización de partes software en prácticamente cualquier fase del ciclo de vida del producto (análisis, diseño e implementación).

Estas partes software, generalmente, se corresponden con fragmentos de código, procedimientos, librerías y programas desarrollados en otros proyectos, y que pueden ser utilizados de nuevo para ser incorporados en ciertas partes del nuevo producto que hay que desarrollar. Además, en estos últimos años se ha podido constatar un aumento en el uso de componentes comerciales en prácticas de reutilización de software. Concretamente, estos componentes comerciales, que comúnmente se conocen con el nombre de componentes COTS (del inglés "*Commercial Off-The-Shelf*"), están siendo considerados con mayor asiduidad para la construcción de sistemas complejos, distribuidos y abiertos. Suelen ser componentes ampliamente extendidos, cuyo precio es significativamente inferior al precio de componentes especialmente diseñados con características similares [Iribarne03].

Para la elaboración de estos sistemas, los ingenieros utilizan metodologías, procesos y técnicas de desarrollo basados en componentes. Por tanto, el sistema a desarrollar estará compuesto por una o más aplicaciones software, que pueden ser consideradas o no como componentes. Incluso puede que algunas de estas aplicaciones software hayan sido construidas mediante la composición de otras partes software (componentes) durante el desarrollo del sistema.

5.2 COMPONENTES COMERCIALES (COTS)

El término COTS, como sucede con muchos otros términos en el campo de la informática, surge desde el Ministerio de Defensa de los Estados Unidos [Oberndorf97]. Históricamente hablando, el término COTS se remonta al primer lustro de los años 90, cuando en Junio de 1994 el Secretario de Defensa americano, William Perry, ordenó hacer el máximo uso posible de especificaciones y estándares comerciales en la adquisición de productos (hardware y software) para el Ministerio de Defensa. En Noviembre de 1994, el Vicesecretario de Defensa para la Adquisición y Tecnología, Paul Kaminski, ordenó utilizar estándares y especificaciones de sistemas abiertos como una norma extendida para la adquisición de sistemas electrónicos de defensa.

El término componente comercial puede ser referido de muy diversas formas, como por ejemplo, software "*Commercial Off-The-Shelf*" (***COTS***), o "*Non-Developmental Item*" (***NDI***), o incluso "*Modifiable Off-The-Shelf*" (***MOTS***) [Carney00]. En realidad existen pequeñas diferencias entre ellos, diferencias que vienen reflejadas en la tabla 5.1. En cualquier caso, se hará referencia a las tres categorías como componentes COTS:

Tabla 5.1: Tipos de software comercial

Categoría	Descripción
COTS	Software que (a) existe a priori, posiblemente en repositorios; (b) está disponible al público en general; y (e) puede ser comprado o alquilado.
NDI	Software desarrollado inicialmente sin un interés comercial por unas organizaciones para cubrir ciertas necesidades internas, y que puede ser requerido por otras organizaciones. Por tanto es un software que (a) existe también a priori, aunque no necesariamente en repositorios conocidos; (b) está disponible, aunque no necesariamente al público en general; y (c) puede ser adquirido, aunque más bien por contrato.
MOTS	Un tipo de software *Off-The-Shelf* donde se permite tener acceso a una parte del código del componente, a diferencia del componente COTS, cuya naturaleza es de caja negra, adquirido en formato binario, y sin tener posibilidad de acceder al código fuente.

Como sucede para el caso de los componentes software, en la literatura tampoco existe una definición concreta y comúnmente utilizada para el término COTS. Una definición híbrida del término *componente COTS* puede ser encontrada en [Brown98]. Un elemento COTS se refiere a un tipo particular de componente software, probado y validado, caracterizado por ser una entidad comercial, normalmente de grano grueso[3], que reside en repositorios software y que es adquirido mediante compra o alquiler con licencia, para ser probado, validado e integrado por usuarios de sistemas.

[3] La noción de componente puede variar dependiendo del nivel de detalle desde donde se mire, conocido como "granularidad". Un componente con granularidad gruesa se refiere a que puede estar compuesto por un conjunto de componentes, o ser una aplicación para construir otras aplicaciones o sistemas a gran escala.

Otra definición de componente COTS podría ser: Un componente COTS es una unidad de elemento software en formato binario, utilizada para la composición de sistemas de software basados en componentes, que generalmente es de grano grueso, que puede ser adquirido mediante compra, licencia, o ser un software de dominio público, y con una especificación bien definida que reside en repositorios conocidos.

Aunque es cierto que desde 1994 se están llevando a cabo prácticas para la utilización de componentes comerciales en procesos de desarrollo, la realidad es que muchas organizaciones encuentran que el uso de componentes COTS conlleva un alto riesgo y esfuerzo de desarrollo, y encuentran problemas para controlar su evolución y mantenimiento dentro del sistema [Dean97]. Estos problemas se deben en cierta medida, a que las organizaciones utilizan procesos y técnicas tradicionales para el desarrollo basado en componentes, pero no para componentes comerciales.

Otro inconveniente es que los fabricantes de componentes COTS tampoco documentan de forma adecuada sus productos para que puedan ser consultados por usuarios desarrolladores que necesitan conocer detalles de especificación del componente, como información acerca de sus interfaces, protocolos de comunicación, características de implantación (tipos de sistemas operativos y procesadores donde funciona, lenguaje utilizado, dependencias con otros programas, etc.) y propiedades funcionales.

Por otro lado, el uso de componentes software COTS tiene problemas serios de certificación (ingeniería inversa), debido a que el proceso de diseño de éstos suele ser desconocido. Además, el software COTS está compuesto por componentes de propósito general y suele poseer pobres especificaciones con respecto a la confiabilidad [IEEE97]. Puesto que los componentes COTS no se diseñan especialmente para el sistema en el que serán integrados, existe un riesgo residual respecto a aquellas partes del componente no utilizadas por el sistema. También se tiene que tener en cuenta que los componentes COTS tienen un tiempo de vida muy corto, debido a que sufren de continuas actualizaciones y modificaciones, y por tanto tampoco pueden beneficiarse de las estadísticas que definirían su tolerancia a fallos.

5.2.1 Características de un componente comercial

Por regla general, existe una gran diversidad de parámetros que caracterizan a un componente COTS, pero sin embargo, dos son los más comunes en la literatura referente a este tipo de componentes. En primer lugar, un componente COTS suele ser de grano grueso y de naturaleza de *caja negra* sin posibilidad de ser modificado o tener acceso al código fuente. Una de las ventajas de un software comercial es precisamente que se desarrolla con la idea de que va a ser aceptado como es, sin permitir modificaciones. Hay algunos desarrolladores de componentes que permiten la posibilidad de soportar técnicas de personalización que no requieren una modificación del código fuente, por ejemplo mediante el uso de *plug-ins* y *scripts*.

En segundo lugar, un componente COTS puede ser instalado en distintos lugares y por distintas organizaciones, sin que ninguna de ellas tenga el completo control sobre la evolución del componente software. Es sólo el vendedor de componentes COTS quien decide su evolución y venta. Son muy numerosas las ventajas, aunque también lo son los inconvenientes al utilizar componentes COTS en lugar de componentes de fabricación propia.

Una de las ventajas más claras es el factor económico, relacionado con el coste de desarrollo. Puede ser mucho más barato comprar un producto comercial, donde el coste de desarrollo ha sido amortizado por muchos clientes, que intentar desarrollar una nueva pieza software. Otra ventaja es que el uso de un producto comercial permite integrar nuevas tecnologías y nuevos estándares más fácilmente y rápidamente que si se construye por la propia organización.

Pero el uso de componentes COTS también tiene algunas desventajas. Destacan principalmente dos, aunque estas derivan en otras más. En primer lugar, los desarrolladores que han adquirido un componente comercial, normalmente no tienen posibilidad de acceso al código fuente para modificar la funcionalidad del componente. Esto significa que en las fases de análisis, diseño, implementación y pruebas, el componente es tratado como un componente de caja negra, y esto puede acarrear ciertos inconvenientes para el desarrollador, como por ejemplo no saber como detectar y proceder en caso de fallos; o que el sistema requiera un nivel de seguridad no disponible en el componente, entre otros problemas. Además, los productos comerciales están en continua evolución, incorporando el fabricante nuevas mejoras al producto y ofreciéndoselo a sus clientes (por contrato, licencia o libre distribución). Sin embargo, de cara al cliente/desarrollador, reemplazar un componente por uno actualizado puede ser una tarea laboriosa e intensiva: el componente y el sistema deben pasar de nuevo unas pruebas (en el lado cliente) que certifiquen la fiabilidad de los mismos integrados en el sistema a desarrollar.

En segundo lugar, otra gran desventaja es que, por regla general, los componentes COTS no suelen tener asociados ninguna especificación de sus interfaces, ni de comportamiento, de los protocolos de interacción con otros componentes, de los atributos de calidad del servicio, y otras características que lo identifiquen. En algunos casos, las especificaciones que ofrece el fabricante de componentes COTS puede que no sean siempre correctas, o que sean incompletas, o que no sigan una forma estándar para describirlas (las especificaciones). Otras veces, aunque el vendedor de componentes COTS proporcione una descripción funcional del componente, puede que ésta no satisfaga las necesidades del integrador, y que necesite conocer más detalles de la especificación del comportamiento y de los requisitos del mismo. Además, la falta de una información de especificación puede acarrear ciertos problemas al desarrollador que utiliza el componente COTS, como por ejemplo la imposibilidad de estudiar la compatibilidad, la interoperabilidad o la trazabilidad de los componentes durante el desarrollo del sistema.

5.2.2. Selección de componentes COTS

La selección de componentes es un proceso que determina qué componentes ya desarrollados pueden ser utilizados. Existen dos fases en la selección de éstos: la fase de búsqueda y la fase de evaluación. En la fase de búsqueda se identifican las propiedades de un componente, como por ejemplo, la funcionalidad del mismo (qué servicios proporciona) y otros aspectos relativos a su interfaz (como el uso de estándares), aspectos de calidad, como son la fiabilidad, la predicibilidad, la utilidad, o la certificación [Voas98], que son difíciles de aislar, y aspectos no técnicos, como la cuota de mercado de un vendedor o el grado de madurez del componente dentro de la organización. La fase de búsqueda es un proceso tedioso, donde hay mucha información difícil de cuantificar, y en algunos casos, difícil de obtener. En la fase de verificación y validación de la integración del componente en el sistema, también nos encontramos ante un serio problema de cómo abordar el aspecto de la confiabilidad del componente a adquirir.

Los sistemas con componentes COTS se construyen mediante la integración a gran escala de componentes adquiridos a terceras partes. La naturaleza de la caja negra de estos

componentes, la posibilidad de que exista una incompatibilidad con la arquitectura, y su corto ciclo de vida, requiere una aproximación de desarrollo diferente [Basili01]. Además, la integración de componentes COTS en el corazón de los sistemas críticos es una cuestión todavía no resuelta. En [IEEE97] se han propuesto algunas soluciones que dependen del atributo de confiabilidad que es considerado. Si se habla de la integridad, la idea es proteger los componentes críticos de un sistema por medio de la inserción de módulos especializados, llamados cortafuegos, dedicados a monitorizar el sistema, éstos están a cargo de controlar las interacciones entre componentes COTS críticos y no críticos.

Con respecto a la seguridad, la inserción de COTS aumenta el problema de los canales secretos. Por ejemplo, un componente de software incluso podría emitir una llamada ilícita a priori desconocida. Una solución conveniente consistiría en filtrar las salidas de la aplicación (noción de envolturas, del inglés "*wrappers*") para prevenir llamadas ilegales. Otra aproximación consiste en establecer una interfaz estándar para los COTS. Tal interfaz normalizada, nos permitiría definir un conjunto de propiedades formales relacionadas a la privacidad y/o seguridad, así como la integración de mecanismos para garantizar la no violación de estas propiedades. En [Voas98] un sistema es considerado como un conjunto de componentes COTS independientes considerados como cajas negras; se propone evaluar la habilidad de un error para propagarse de un componente a otro por medio de su interfaz.

Además, actualmente en el campo de los sistemas *on-chip* (SoCs del inglés "*Systems-on-a-Chip"*), computadores empotrados en un chip, se constata que estos dispositivos cada día tiene más capacidad para realizar tareas mucho más complejas, debido a que su arquitectura interna crece acorde a los tiempos y hace que sean capaces de ejecutar software cada vez más sofisticado. La creciente complejidad del software que estos sistemas son capaces de ejecutar junto con la necesidad de reducir el tiempo en el que un producto debe estar disponible en el mercado hace que los fabricantes de este tipo de sistemas acudan cada vez más a reutilizar componentes COTS. Pero la heterogeneidad de este tipo de componentes ha motivado la necesidad de verificar su comportamiento en este tipo de sistemas en ausencia y en presencia de fallos.

En este sentido, las técnicas de inyección de fallos se constituyen como técnicas fundamentales para la evaluación de la robustez de este tipo de componentes. Básicamente estas técnicas establecen procesos para emular fallos en sistemas informáticos y monitorizar el comportamiento resultante [Madeira00]. Desde el punto de vista de la evaluación de la robustez de los componentes de sistemas empotrados en chip, la mayoría de las técnicas de inyección de fallos aproximan el problema desde una perspectiva hardware. Pero como se dice en [Kanoun05] éstas no suelen ser portables y normalmente requieren de modificaciones importantes en el sistema bajo estudio además de introducir una sobrecarga temporal significativa en la mayoría de los casos, es por ello que se hace necesario un enfoque desde la perspectiva del software para evaluar dicha robustez de los componentes COTS.

5.2.3 Robustez de componentes COTS

Los sistemas críticos de tiempo real en sistemas empotrados normalmente son verificados y testados antes de su aparición en el mercado o utilización como productos finales, pero aún a pesar de esto suelen aparecer alrededor de 10 defectos de software por cada 1000 líneas de código [Regan04]. Este tipo de defectos del software es lo que denominamos como fallos residuales del software, fallos que no son detectados en fases previas del testeo del producto.

Para ver la idoneidad en la elección de unos componentes frente a otros, por el hecho de que puedan aparecer dichos fallos residuales en la integración de los mismos en el desarrollo final

del sistema, es necesario realizar una serie tests que justifiquen la elección de los componentes. Para ello una de las posibilidades que tradicionalmente se han venido utilizado es la de realizar tests de robustez del software.

Así, definimos en primer lugar la robustez de un componente como el grado en el que ese componente trabaja de forma correcta en presencia de valores de entrada excepcionales o condiciones de entorno estresantes. La robustez puede ser vista como un indicador de la capacidad del componente para resistir o reaccionar frente a fallos inducidos por las aplicaciones que funcionan en capas superiores o fallos que pudieran venir de *drivers* de dispositivos o más genéricamente de capas del hardware [Dbench04].

La robustez es un atributo clave de los sistemas empotrados que tiene un importante impacto en la confiabilidad final del sistema. Los tests robustez ya han sido utilizados en otras ocasiones para evaluar componentes COTS, revelando las deficiencias en cuanto a confiabilidad de los mismos, sobre todo cuando la estructura interna de los componentes COTS es desconocida y no se dispone de su código fuente [Laplante97], [Rodríguez02], [Moreira03] y [DOT/FAA02]. Es por ello que en los tests de robustez normalmente se considera al sistema/componente como una caja negra [ISO/IEC 61508] y la inyección de fallos se desarrolla sobre los valores de los parámetros de su interfaz.

Dentro de los diferentes componentes COTS que podríamos incorporar en un sistema encontramos los sistemas operativos de tiempo real (del inglés *RTOS,"Real-Time Operating System"*). Así definimos a un sistema operativo de tiempo real como un software que maneja la temporización en un microprocesador o microcontrolador de forma cuidadosa. Los sistemas de tiempo real se caracterizan por el hecho de que si no se conjugan correctamente tanto las operaciones lógicas como las de tiempo, el sistema puede fallar de forma catastrófica. Así, los tests de robustez sobre sistemas operativos de tiempo real pueden ser muy útiles para además obtener medidas que caractericen el comportamiento temporal del sistema operativo en presencia de fallos.

Los sistemas operativos de tiempo real cada vez son más utilizados en sistemas empotrados, y las aplicaciones que funcionan sobre éstos dependen de los servicios que el sistema operativo sea capaz de ofrecer, por un lado de modo correcto pero por el otro, respetando las restricciones de tiempo que los requisitos o especificaciones funcionales del sistema establecen. Para ello, en el modelo de fallo definido para realizar los tests de robustez se aplican valores erróneos en los parámetros de las llamadas al sistema. Este tipo de fallos son ampliamente utilizados tanto en los tests de robustez como los tests de interfaz de componentes [Kropp98].

En este trabajo que se presenta se pone especial énfasis, por todo lo dicho anteriormente, en cómo evaluar la respuesta de sistemas operativos de tiempo real frente a comportamientos erróneos de las aplicaciones que funcionan en capas superiores, es decir frente a fallos residuales de los componentes, aunque la metodología que se propone perfectamente puede ser trasladada a otras aplicaciones de sistemas empotrados que no hagan uso de RTOS. En definitiva y como posteriormente se explicará, se propone una metodología de evaluación de la robustez de componentes COTS utilizando técnicas de inyección de fallos.

5.3 METODOLOGÍA DE INYECCIÓN DE FALLOS

Una vez se ha establecido que lo que interesa es emular fallos residuales del software en componentes COTS, y que para ello se van a llevar a cabo tests de robustez sobre la interfaz de los mismos. A continuación se va a describir la metodología propuesta para la realización de la inyección de fallos en los parámetros de las llamadas a un componente COTS. En este caso para

presentar la metodología se ha escogido como ejemplo de componente un sistema operativo de tiempo real.

En esta sección se propone una nueva metodología de evaluación de la robustez de componentes COTS en sistemas empotrados *on-chip* desde la perspectiva del software. Cuestiones importantes que se plantean serán cómo emular los fallos residuales externos y cómo estudiar la robustez de los componentes frente a ese tipo de fallos. Es bien conocido que el creciente incremento en la utilización de componentes COTS en el desarrollo de sistemas empotrados está aumentando la probabilidad de fallos software de tipo residual. Además debido a la integración y la interacción de dichos componentes, las consecuencias de la activación de estos fallos residuales pueden propagarse de unos componentes a otros en forma de errores. Es por ello que la evaluación de la robustez de los componentes frente a fallos externos es muy importante en la elección de componentes para sistemas empotrados.

Sin embargo, y debido al alto nivel de encapsulamiento que se da hoy día en los Sistemas-on-Chip la emulación de cualquier tipo de fallo es todo un reto. Éste es un problema conocido y expuesto por algunos autores [Yuste03c][Skarin04] que ya han intentado resolver este inconveniente a través de la utilización de los mecanismos de depuración *on-chip* que ofrecen muchos de los actuales sistemas empotrados.

A priori, la solución que se debería proponer debe tener las siguientes características:

- Portabilidad: la solución propuesta debe considerar características de depuración *on-chip* que sean estándar.

- No intrusión: tanto la emulación del fallo como el proceso de monitorización no deben requerir modificaciones en el sistema bajo estudio. Además, la sobrecarga temporal debe ser mínima o inexistente puesto que el objetivo es evaluar también la robustez de componentes de tiempo real.

- La solución debe permitir inyectar fallos bajo condiciones de funcionamiento reales, lo que significa que no se debe utilizar modelos para simular el entorno de ejecución ni el proceso físico bajo control. Esto es importante para:

    - Sistemas-on-Chip ejecutando software dedicado al control de procesos físicos cuya complejidad hace imposible definir modelos precisos y detallados.

    - Sistemas empotrados con restricciones de seguridad importante en donde la valoración de la robustez bajo condiciones reales es en ocasiones obligatorio para la certificación de los mismos [ISO/IEC 61508].

- Y por último la solución propuesta debe poder ser aplicada a cualquier componente funcionando sobre sistemas empotrados *on-chip* donde el código fuente de dichos componentes no está disponible.

5.3.1 Fallos residuales del software y sistemas on-Chip

Como se ha comentado anteriormente, los fallos residuales del software pueden ser vistos como pequeños defectos en componentes que no fueron detectados durante las fases previas de testeo o fases de evaluación. Es importante señalar que el objetivo no es sólo emular la ocurrencia de tal tipo de fallos en un componente en particular, sino que más bien el objetivo es emular la propagación de sus consecuencias de un componente a otro.

Para realizar tal emulación debe considerarse que cada componente tiene una interfaz pública que centraliza la recepción de todos los mensajes de llegada. Esta interfaz es por tanto el conducto a través del cual los errores manifestados pueden propagarse de un componente a otro. Por tanto, la propagación de estos errores puede ser emulada a través de la corrupción de los valores de los parámetros que un componente envía a otro componente.

Esto llevado a la práctica supone que corromper el valor de un parámetro es sinónimo de modificar los registros internos del sistema *on-chip*. Para comprender este punto, necesitamos saber que, debido a consideraciones de rendimiento, la mayoría de los compiladores para sistemas empotrados utilizan los registros internos del sistema con el objetivo de realizar la carga de los parámetros durante las llamadas al componente. Esto permite optimizar la velocidad de ejecución de las aplicaciones, aunque desafortunadamente también significa que para modificar el contenido de los registros, que albergan el valor de los parámetros de las llamadas al componente, se debe parar la ejecución, por la imposibilidad de modificar el contenido de los mismos *on-the-fly* (sin parar la ejecución) que la clase 3 de Nexus™ en este caso nos ofrece. Esto haría que para la corrupción de los parámetros se deba detener momentáneamente el funcionamiento del sistema y hace que el proceso de inyección al final sea intrusivo. Acorde al estado del arte de las técnicas actuales, la corrupción del contenido de los registros internos del sistema se ha hecho desde siempre parando la ejecución del sistema bajo evaluación, pero como se ha dicho anteriormente si se quiere evaluar a los sistemas de un modo más realista se debe evitar esto. Es por ello que como objetivo de este trabajo se debe hallar una forma de realizar la corrupción de los parámetros de las llamadas al componente sin detener la ejecución del sistema.

5.3.2 Emulación de fallos software a través de depuración on-Chip

El objetivo de esta sección es explicar cómo la corrupción del contenido de los registros internos, para modificar el contenido de los parámetros de las llamadas al componente, puede ser llevada a cabo de un modo no intrusivo, y ver cómo el comportamiento del componente bajo estudio puede ser monitorizado después de realizar la inyección del fallo.

La metodología que se propone contempla cinco fases sucesivas [Pardo05a]:

1. En primer lugar hay que determinar dónde pueden ser inyectados los fallos. En este paso se establece un mapeo de memoria entre el punto de vista lógico de los componentes y la zona física de memoria donde se hallan ubicados. Este mapeo requiere de cuatro tareas a ser realizadas:

    a) Seleccionar un componente para la experimentación. Nombrar a este componente como el ***"componente objetivo",*** será la función de la API del componente en la que se va a realizar la inyección de fallos.
    b) Localizar las zonas de memoria donde están localizadas las funciones de la interfaz del componente objetivo. Llamar a estas direcciones como ***"direcciones del componente invocado",*** la correspondencia de direcciones se obtiene del mapa de memoria generado por el linker (o enlazador).
    c) Determinar dónde otros componentes del sistema invocan funciones de las direcciones del componente invocado. Llamar a estas direcciones como ***"direcciones de invocación";*** éstas contienen la instrucción de salto al código del componente seleccionado.
    d) Fijar la atención en las direcciones de memoria precedentes a esas direcciones de invocación. Estas direcciones se corresponden con aquellas que codifican el almacenamiento de los valores de los parámetros de las

llamadas a los componentes en los registros internos del sistema. Éstas implementan el intercambio de parámetros entre el componente que llama y el componente que es llamado. Por tanto, los valores almacenados en estas direcciones serán aquellos propuestos para realizar la inyección de fallos. A estas direcciones de memoria se las denomina como ***"direcciones potencialmente candidatas para la inyección".***

2. Elegir la localización del fallo y su emulación:

    a) De entre todas las direcciones potencialmente candidatas para la inyección seleccionar una y denominarla como ***"dirección para la inyección"*** del experimento.
    b) Definir la carga de trabajo a ser ejecutada en el sistema. La carga variará de un sistema a otro, dependiendo del proceso físico o de control que se quiera implementar.
    c) Lanzar a ejecución al sistema y definir un disparo para lanzar el proceso de emulación del fallo (o inyección del error).
    d) Determinar cuál es el valor que contiene el parámetro de la dirección en la que se va a llevar a cabo la inyección. Entonces modificarlo utilizando los mecanismos de lectura/escritura al vuelo (del inglés "*on-the-fly*") que muchas de las interfaces de depuración *on-chip* proveen como mecanismos de calibración. Estos mecanismos permiten modificar el contenido de zonas de memoria sin perturbar la ejecución del sistema. En este sentido, el contenido que va a ser almacenado en los registros es cambiado utilizando valores modificados durante el intercambio de parámetros.

3. Monitorizar el comportamiento de la dirección del componente invocado en presencia de los fallos emulados. Esto debe ser llevado a cabo a través de una perspectiva de caja negra, es decir, se deben observar sólo las salidas del componente llamado. A los resultados observados se los denomina como ***"traza del sistema en presencia de fallos".***

4. Lanzar una ejecución libre de fallos denominada como "***Golden-run***": En este caso no se debe inyectar fallo alguno y se debe observar cómo se comporta el sistema. Esta *golden-run* es la ejecución de referencia que nos dice cómo debería funcionar el sistema en ausencia de fallos. A la traza de observación de dicha ejecución la denominaremos "***traza del sistema libre de fallos***".

5. Comparar el conjunto de observaciones de las trazas de ejecución en presencia y libres de fallos. Las diferencias encontradas en la comparación de ambas marcará un comportamiento erróneo del sistema frente a los fallos inducidos. Esto es obvio que será cierto siempre que consideremos para ambas ejecuciones condiciones de funcionamiento y activación equivalentes.

A continuación se va a detallar más cada una de los pasos a seguir en la metodología que se propone.

5.3.2.1 Paso 1: Determinar dónde pueden ser inyectados los fallos.

Tal como se ha comentado anteriormente, las aplicaciones en SoCs pueden verse como a agregaciones de componentes que interactúan en tiempo real a través de sus interfaces. Este punto de vista tan abstracto contrasta con la visión a bajo nivel que se obtiene cuando observamos la actividad de estos componentes utilizando los mecanismos de depuración

provistos por las interfaces OCD de los sistemas *on-chip*. Por tanto, lo que es necesario es obtener una metodología para correlacionar la visión que se tiene del componente como tal, con su correspondiente mapa de memoria. Este es un paso esencial para determinar con exactitud dónde pueden ser inyectados los fallos durante la experimentación.

Componentes objetivo

En primer lugar lo que se debe realizar es la selección del componente objetivo de la experimentación y obtener de su especificación la definición de su interfaz. Posteriormente una función de su interfaz y un parámetro de esa función deben ser seleccionados para el experimento. Recordar que debido a las restricciones impuestas por considerar un punto de vista de caja negra del sistema, el código fuente del componente se considera que no estará disponible (sin embargo se asume que el fichero de código objeto de cada uno los componentes está accesible).

Dichos ficheros de código objeto contiene el código máquina que define el comportamiento de cada componente. Una vez que un conjunto de componentes del sistema han sido seleccionados, éstos son localizados en la memoria interna del SoC. Como resultado el linker genera un mapa de memoria del SoC. Este mapa reflejará, por tanto, en qué zonas de memoria están localizadas las funciones del componente.

Dirección del componente invocado

En primer lugar decir, que no es muy complicado localizar las funciones del componente invocado utilizando la información que proporcionan los mapas de memoria del SoC. Lo único que al final hay que hacer, es localizar la interfaz de la función del componente seleccionado en el mapa de memoria de la aplicación bajo estudio instalada en el SoC. Esto puede ser llevado a cabo gracias a que las localizaciones de memoria en los ficheros que albergan la información del mapa de memoria, mantienen los nombres de las funciones en alto nivel. La localización de las funciones es por tanto traducida a sus correspondientes direcciones de memoria física cuando el mapa de memoria es transferido al SoC. Así pues, la localización de memoria física del comienzo de la función del componente es lo que se ha llamado como la dirección del componente invocado.

Dirección de invocación y direcciones para la inyección

Una vez que ha sido determinada la dirección del componente invocado, pueden ser localizadas en la memoria del SoC las llamadas a función en otros componentes a esa dirección. Aquellas posiciones que contienen las instrucciones que codifican esas llamadas se han denominado como direcciones de invocación.

Por tanto, para poder localizar las direcciones de invocación el mapa de memoria de la aplicación debe ser analizado. El objetivo de este análisis es encontrar las llamadas a función de las direcciones del componente invocado seleccionadas. Para llevar a cabo esto, se debe saber cuál es el patrón que el compilador sigue para codificar una llamada a función. El compilador puede utilizar la memoria global, la pila o los registros internos para llevar a cabo el pase de parámetros en las llamadas a función. En este caso el interés se centra en el pase de parámetros a través de los registros internos del micro puesto que en muchos SoCs éstos son utilizados en el pase de parámetros con la finalidad de optimizar la velocidad de ejecución de las aplicaciones. El problema, y como se ha comentado en varias ocasiones, es que la corrupción del valor que albergan los registros internos no puede ser llevada a cabo sin parar la ejecución del sistema, y es por ello el interés por encontrar una solución diferente. Debido a la naturaleza intrínseca de los compiladores esta clase información está bien establecida y documentada, y debe ser conocida.

En general la estrategia seguida por los compiladores para llevar a cabo el intercambio de parámetros pueden caracterizarse por:

- Un conjunto de instrucciones en código máquina donde los valores de los parámetros son copiados a los registros internos del SoC.

- Una instrucción que contiene un salto a la dirección del componente invocado, como por ejemplo puede ser la localización de una llamada función.

- Y en caso de retorno de valores, las instrucciones que codifican el retorno de tales valores.

Así por tanto, una vez que el patrón seguido por el compilador es identificado, se puede automatizar el proceso de obtención del conjunto de direcciones donde lo valores de los parámetros son manipulados. Éstas serán entonces, las direcciones donde se debe realizar la inyección de fallos.

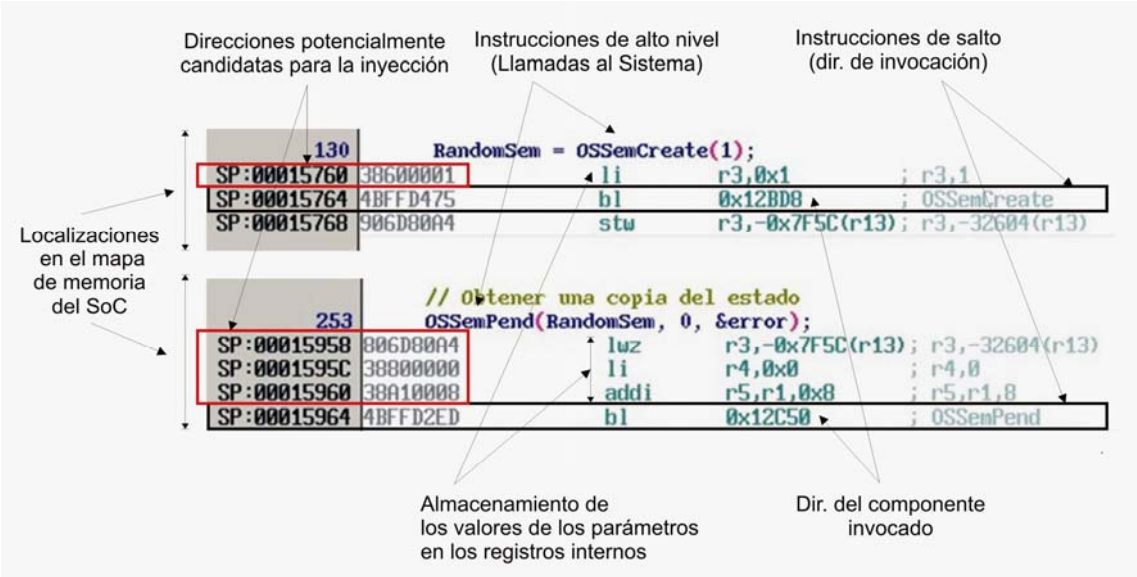


Figura 5.1: Localizaciones en memoria, direcciones para invocación e inyección

La figura 5.1 ilustra todo el análisis realizado para la localización de las direcciones de interés donde llevar a cabo la inyección de fallos. Como se puede observar en este extracto del código desensamblado correspondiente a una aplicación ejemplo, se tienen dos llamadas al sistema candidatas la para inyección de fallos, una es "*OSSemCreate*" que posee un sólo parámetro y la otra es "*OSSemPend*" que posee tres parámetros. Como se puede observar en la figura, ambas llamadas siguen un mismo patrón para el pase de parámetros de las mismas. Dicho patrón al que se hace referencia consiste en que siempre se realiza en primer lugar una carga de los parámetros de la llamada al componente en los registros internos del microcontrolador y posteriormente se realiza un salto a la dirección donde se halla el código fuente de la llamada al componente correspondiente. Tomando como ejemplo la llamada al sistema "*OSSemCreate*" en la figura 5.1 se puede observar que en la dirección de memoria "0x15760", correspondiente al mapa de memoria del SoC, existe una instrucción que carga el valor del parámetro (en este caso igual a 1) de la función de llamada en el registro interno "r3" del microcontrolador. Posteriormente, como se puede ver, existe una instrucción de salto a la dirección "0x12BD8", que se corresponde con la dirección de memoria donde se halla ubicado el código fuente que implementa dicha llamada al sistema. Para el caso de la llamada "*OSSemPend*" habrá tres instrucciones de carga correspondientes a los tres parámetros que implementa dicha llamada.

Dentro de lo que se ha llamado como localizaciones en el mapa de memoria del SoC estarán las direcciones potencialmente candidatas para la inyección, donde dentro del contenido de esas direcciones en la parte baja, se codifica el dato que debe ser insertado y en la parte alta la instrucción y registro especificados. Esta secuencia de acciones a las que se ha denominado como patrón, y que posteriormente se verá que se ha verificado con distintas aplicaciones, permite establecer un algoritmo de análisis para la obtención de las direcciones de memoria que se corresponde con la carga de los parámetros de las llamadas al componente.

5.3.2.2 Paso 2: Elección de la localización y emulación del fallo.

Una vez que todas las potenciales direcciones para llevar a cabo la inyección han sido determinadas, es hora de elegir una para realizar el experimento. Dicha selección puede ser llevada a cabo de forma determinista o siguiendo una aproximación aleatoria. En este caso se ha optado por una selección de las direcciones a inyectar de forma aleatoria.

El contenido de las direcciones para la inyección seleccionadas debe ser analizado con el propósito de la emulación del fallo. Básicamente esta dirección de memoria contiene una instrucción con tres campos: un código de instrucción de carga (load), un registro interno del sistema donde almacenar el valor y el parámetro de la función. La distribución y el número de bytes asociados a cada uno de estos campos, depende del tipo de arquitectura considerada para el SoC. En este caso los parámetros de la función son codificados en las partes bajas de la palabra de la instrucción y esos serán los bits que se deben cambiar para corromper el pase de parámetros. En ambos casos, los valores de los parámetros son corrompidos antes de que éstos sean cargados en los registros internos del sistema, de este modo es posible utilizar las capacidades de los mecanismos de depuración de lectura/escritura *on-the-fly* de que dispone el SoC, y así llevar a cabo la corrupción del parámetro en cuestión. Esta funcionalidad del sistema de depuración trabaja en paralelo con el resto de la circuitería interna del SoC y no interfiere en la ejecución final del sistema.

La corrupción de los parámetros es llevada a cabo en tres etapas sucesivas. En primer lugar es leído el contenido de la instrucción que alberga la dirección para inyección. Posteriormente el valor del parámetro que contiene dicha instrucción es corrompido. Finalmente una versión modificada de la instrucción es reescrita en memoria. Con el objetivo de determinar el valor modificado a inyectar en el parámetro de la función existen dos posibilidades o aproximaciones diferentes. La primera consiste en realizar un ***"bit-flip"*** de uno de los bits en la representación binaria del valor. La segunda aproximación denominada como "***selección sustitutiva"***, se sustituye el valor del parámetro por otro que sea inválido, se salga de rango u otras opciones que se consideren como posibles valores críticos. En este caso el nuevo valor debe ser seleccionado en función del tipo de datos del parámetro que se va a modificar. Sin embargo estudios han demostrado la equivalencia de los errores provocados por ambas técnicas [Yuste03c], por ello se ha optado por la técnica del *bit-flip* por ser más fácilmente implementable.

Aunque interesante desde el punto de vista de la inyección de fallos, los mecanismos OCD tienen capacidades limitadas en cuanto a la sincronización del momento de la inyección con la ejecución del componente bajo estudio. Una posibilidad es adoptar una estrategia basada en la inyección a partir de la ejecución de una instrucción particular en memoria. Sin embargo, es difícil asegurar que cada inyección sea llevada a cabo antes de que otra instrucción, como pudiera ser la instrucción que contiene la dirección para inyección, sea ejecutada. Este aspecto que no es relevante cuando el sistema es detenido para realizar la inyección de fallos, llega a ser un reto importante cuando uno de los requerimientos es la no intrusión. En este caso se han utilizado temporizadores del sistema que son inicializados con el inicio de la ejecución del mismo y cargados con un valor aleatorio elegido dentro del intervalo de tiempo especificado

para el experimento. En definitiva la herramienta de inyección espera a que se alcance un tiempo determinado para llevar a cabo el proceso de inyección del fallo.

5.3.2.3 Paso 3: Monitorización del componente bajo estudio.

Monitorizar la actividad de las aplicaciones funcionando en SoCs utilizando los mecanismos e interfaces de depuración OCD tampoco es muy complicado. La idea consiste en explotar las capacidades de traza que el OCD ofrece, para observar qué es lo que está sucediendo en el interior del SoC.

En este caso las observaciones de interés se focalizan en los parámetros de salida de la interfaz del componente bajo estudio y sus valores de retorno. La idea es seleccionar posiciones de memoria donde tales valores son albergados y programar la interfaz de depuración OCD para seguir su evolución. Cada vez que una lectura o acceso a escritura es llevada a cabo en las posiciones seleccionadas, la circuitería interna del OCD genera un mensaje de traza identificando el tipo de acceso y el valor que ha aparecido.

Las excepciones hardware también son una fuente de información importante a ser monitorizada, ya que éstas denotan situaciones conflictivas derivadas de la incapacidad de los componentes para exhibir un comportamiento robusto en presencia de los fallos emulados. Debido a que las capacidades de detección del sistema no son infalibles, cierta cantidad de fallos emulados podrían propagarse al entorno de ejecución de otros componentes y llevar al sistema a un estado de "*cuelgue*" o funcionamiento incorrecto. Para detectar tales situaciones, el dispositivo que controla desde el exterior los experimentos debe monitorizar al sistema y ser capaz de detectar como éste reacciona ante los nuevos valores de entrada de los parámetros de las funciones.

5.3.2.4 Paso 4: Ejecución de la Golden-run.

Cada experimento en inyección de fallos debe ser seguido o precedido, el orden no es importante, por la ejecución de una *golden-run* o ejecución de referencia libre de fallos. Dicha ejecución libre de fallos debe hacer funcionar al sistema bajo las mismas condiciones operacionales que los experimentos en los cuales se ha inyectado un fallo. La diferencia está en que en la *golden-run* los fallos no son emulados. Las trazas obtenidas en este caso serán trazas libres de errores que definirán la referencia para el análisis posterior de los diferentes experimentos en inyección de fallos y así tener la posibilidad de poder comparar los resultados obtenidos con los esperados.

5.3.2.5 Paso 5: Análisis de resultados

En una última instancia para cada uno de los experimentos ambas trazas deben ser comparadas, pero es importante denotar que para que esta comparación sea representativa las condiciones de activación en el sistema deben ser garantizadas para ambas ejecuciones, la libre de fallos y la que el fallo es inyectado.

Esto significa, que bajo condiciones de ejecución equivalentes (configuración del sistema, carga de trabajo, estado de los procesos controlados, etc.) el comportamiento del sistema debe ser el mismo, de este modo los experimentos pueden ser reproducidos. Y por tanto, bajo esas mismas condiciones ambas trazas, la libre de errores y la de inyección de fallos pueden ser comparadas. En definitiva lo ideal es que el comportamiento el sistema sea determinista.

Desde un punto de vista más purista, diremos que dos trazas pueden ser comparadas si los eventos que reflejan son los mismos y han ocurrido en el mismo orden. Si se tienen en cuenta consideraciones temporales entonces tales eventos también deben haber sido llevados a cabo en el mismo instante de tiempo en cada experimento. Este aspecto es importante cuando consideramos componentes de tiempo real con importantes restricciones temporales. En ese caso, se dice que dos trazas son equivalentes si reflejan que los mismos eventos ocurrieron en el mismo orden y cuando son comparados uno a uno hacen referencia al mismo estado del sistema y reflejan el mismo comportamiento con respecto a los *deadlines* establecidos. Basándonos en este criterio los datos provenientes del proceso de análisis reflejarán cuando las trazas obtenidas con y sin fallos son o no equivalentes. En caso de equivalencia, se concluye que el fallo inyectado al final no ha afectado al comportamiento del componente bajo estudio.

5.3.3 Observación del comportamiento temporal

Durante el proceso de obtención de información del sistema, la monitorización del mismo puede perturbar o influenciar el comportamiento final del sistema. Esta perturbación que comúnmente se denomina como ***"perturbación de monitorización"*** (del inglés "*monitoring perturbation"*) es la representación cuantitativa de la perturbación que el sistema de monitorización causa al sistema bajo estudio. [Smaili04] en su estudio determina que una interferencia producida por la actividad de monitorización en un sistema de tiempo real con fuertes restricciones temporales es algo intolerable. Dicha interferencia causada por esta monitorización puede cambiar el comportamiento temporal del sistema bajo estudio. En este caso la información recogida representaría el comportamiento del sistema durante el tiempo ejecución en el nivel de abstracción intencionado, lo que no es compatible con el estado del sistema si éste no hubiera sido observado por un sistema de monitorización.

Como uno de los objetivos importantes de este trabajo es la evaluación de los atributos temporales que muestran los componentes ante la aparición de fallos; durante la ejecución de los experimentos el comportamiento temporal del sistema es monitorizado. Como se ha dicho anteriormente, el proceso de monitorización no debe ser intrusivo, es por ello que se ha utilizado una interfaz de depuración OCD como es Nexus™ para realizar este estudio. Como se ha visto en el capítulo anterior, esta interfaz fue desarrollada con el propósito original de interfaz de depuración, pero como se ha apreciado ya en otros artículos [Ruiz04][Rebaduengo99] la posibilidad de utilización de Nexus™ como interfaz de inyección de fallos queda demostrada. Sus características más importantes relacionadas con este estudio son su portabilidad, la no intrusión en el proceso de depuración y la posibilidad de obtener estimaciones temporales precisas con el objetivo de estudiar sistemas de tiempo real.

La técnica que se presenta trata de explotar todas las capacidades de observación de Nexus™ con el objetivo de construir trazas de ejecución de las aplicaciones y componentes bajo estudio. La traza de datos de Nexus™ ofrece la posibilidad de monitorizar los accesos de lectura/escritura a memoria *on-the-fly*, lo que combinado con los mecanismos de *watchpoints*, permite conocer cuándo un fallo en memoria llega a convertirse en un error [Skarin04]. Estos *watchpoints* son parte de la circuitería de depuración interna de Nexus™ dentro del microcontrolador y son utilizados con el objetivo de señalar eventos que transcurren en la aplicación sin parar la ejecución de la misma. Básicamente Nexus™ ofrece la posibilidad de definir dos tipos de eventos de aplicación: (i) la ejecución de un código de instrucción dado, y (ii) los accesos a una palabra de memoria de datos predeterminada.

Además, los datos obtenidos de la traza de Nexus™ incluyen una huella temporal del flujo de ejecución de la aplicación que funciona en el sistema. Esto hace posible medir el tiempo transcurrido entre diferentes eventos para obtener por ejemplo las latencias de error de un sistema operativo o el tiempo utilizado por una tarea durante su ejecución en un sistema

multitarea. Las trazas de ejecución resultantes representan por tanto, el conocimiento que puede ser obtenido a través de la interfaz Nexus™ de lo que es la actividad interna de la aplicación empotrada en el SoC. Básicamente, esta es la información que será utilizada para analizar el comportamiento del sistema después de un experimento en inyección de fallos.

De entre las diferentes medidas que pueden ser monitorizadas, cabe señalar las latencias de detección de los códigos de error proporcionados por los componentes de interés de estudio. Para tal evaluación es monitorizado el instante en donde el fallo inyectado se hace efectivo, como por ejemplo cuando la dirección de memoria afectada es ejecutada, y el momento cuando el componente a través de sus mecanismos de detección notifica un código de error. Para llevar a cabo esto, dos *watchpoints on-chip* deben ser establecidos para obtener la referencia temporal relativa en la sucesión de ambos eventos. Posteriormente la reconstrucción de la traza de Nexus™ es utilizada para obtener información del tiempo desde que el fallo inyectado se hace efectivo hasta la detección del correspondiente error, si lo ha habido. Los pasos a seguir son las siguientes:

1. Poner un *watchpoint on-chip*, en la dirección donde va a ser inyectado el fallo.
2. Establecer este *watchpoint* como una referencia temporal.
3. Definir otro *watchpoint on-chip* en la dirección de memoria donde el mecanismo de detección de errores del componente bajo estudio informa acerca de los códigos de error.
4. Estudiar la traza Nexus™ para obtener la diferencia temporal en la ocurrencia de ambos *watchpoints* o puntos de observación.

La figura 5.2 muestra un ejemplo de traza del tiempo necesitado por el componente COTS bajo estudio para detectar un error en su interfaz. Como se puede observar en la figura, se ha establecido una referencia temporal en la dirección de memoria “C:0x32F8”. En este ejemplo, dicha dirección se corresponde con el parámetro de una llamada al sistema donde el fallo es inyectado; la ejecución de dicha dirección es monitorizada para establecer una referencia temporal y observar la efectividad de la inyección de fallos. Posteriormente otra referencia temporal en la dirección “D:0x3F9804” es establecida también. Dicha dirección se corresponde con el mecanismo que lleva a cabo la notificación de los correspondientes códigos de error proporcionados por el componente. En este experimento el código de error como puede ser visto es el “4301”, que es el dato que contiene dicha dirección, y el tiempo relativo entre ambos eventos, es decir desde la inyección de un fallo que se hace efectivo a la subsiguiente detección del error por parte del componente en este caso fue de 28,940us.

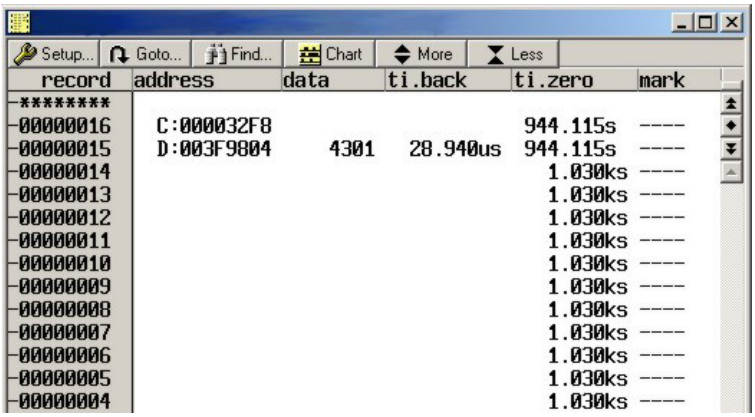


Figura 5.2: Ejemplo de traza Nexus™ para obtener el tiempo requerido por un componente en la detección de error

Una vez se ha visto, que gracias a las características *on-the-fly* de Nexus™ es posible obtener de una forma no intrusiva los tiempos referentes a las latencias de detección de error. Es posible además obtener los tiempos de ejecución de las tareas aplicando el mismo método, lo cual nos ofrece la oportunidad de obtener el comportamiento temporal de las diferentes tareas en el sistema frente a fallos inducidos en la interfaz de los componentes. Para llevar a cabo este análisis los pasos a seguir serán los siguientes:

1. Poner un *watchpoint* en la dirección del comienzo de la tarea.
2. Establecer este *watchpoint* como una referencia temporal de entrada.
3. Definir un *watchpoint* en la dirección final de memoria de la tarea.
4. Estudiar la traza de Nexus™ para obtener la diferencia temporal en la ocurrencia de ambos *watchpoints* o puntos de observación.

Pero debido a las limitaciones de las actuales implementaciones de Nexus™ (dos *watchpoints on-chip* de lectura/escritura y cuatro para instrucciones) [Skarin04], para llevar a cabo la medición temporal de la ejecución de todas las tareas, ha sido utilizado un método alternativo. Dicha alternativa consiste en obtener la traza la ejecución de la aplicación para analizar los rangos de las direcciones de memoria ejecutados. Por tanto, grupos de código fuente han sido definidos para cada tarea con el objetivo de establecer un marco en el sistema para cada uno de los componentes a ser monitorizados. Después de la ejecución y obtención de las diferentes trazas de Nexus™, éstas son filtradas para obtener datos acerca de mínimos, máximos, tiempos promedios, etc. en relación con el comportamiento temporal de los diferentes componentes. Además, haciendo uso del conocimiento de las estructuras de datos internas del componente en cuestión y por la posibilidad que nos ofrece el mapa de memoria de obtener los símbolos correspondientes a los nombres de las diferentes tareas del sistema, es posible conocer cada uno de los estados por los que las diferentes tareas del sistema pasan durante el tiempo de ejecución del experimento.

Con estos datos puede ser obtenida la influencia de los errores en los tiempos de ejecución y así entender cuáles son los límites temporales o *deadlines* de las tareas para que éstas puedan llevar a cabo su trabajo de un modo correcto, aún a pesar de la aparición de errores, o por otro lado observar cómo los errores pueden provocar importantes retardos en la entrega del servicio. Además es posible verificar si las tareas están cumpliendo o no las restricciones temporales que vienen impuestas por los requisitos del sistema.

Es de considerar que tal evaluación temporal puede ofrecer grandes posibilidades a los diseñadores de aplicaciones de sistemas empotrados de tiempo real. En este caso el diseñador podría desarrollar tareas de recuperación del sistema y medir sus tiempos de respuesta como si éstas fueran labores llevadas a cabo en el sistema como otras tareas más. Además, como el diseñador podría conocer los tiempos de latencia de detección de errores, éste podría analizar la planificación de las tareas de un modo teórico con los datos temporales obtenidos o de un modo completamente experimental con el sistema trabajando en tiempo real para obtener la totalidad del comportamiento temporal del sistema y sus diferentes componentes.

5.4 RESUMEN Y CONCLUSIONES

En este capítulo se ha hablado de cómo los componentes COTS cada día son más utilizados en el desarrollo final de los sistemas, a través de la integración de los mismos para un desarrollo más acelerado y con un coste asociado menor. Se han mostrado las distintas definiciones de lo que se entiende por componente COTS y cuales son sus orígenes para poder establecer el marco

en el que se mueve esta tesis. También se ha visto cómo la utilización de éstos, integrados con otros componentes que pudieran haber sido desarrollados por otros fabricantes, puede acarrear problemas de robustez y como se hace necesaria una evaluación de los mismos en este sentido. Para ello, desde la bibliografía se ha propuesto las técnicas de inyección de fallos como medio para evaluar la robustez de los componentes, la problemática asociada a la implementación de las diferentes técnicas y como esto se agrava si se habla de sistemas empotrados de tiempo real. Al final se ha propuesto un metodología para abordar el problema presentado especificando punto a punto los pasos a seguir para realizar la inyección de fallos de diseño del software, y así obtener una evaluación de la robustez y comportamiento temporal de componentes COTS integrados en sistemas empotrados para aplicaciones de tiempo real.

Capítulo 6

DESCRIPCIÓN DEL SISTEMA EXPERIMENTAL

6.1 INTRODUCCIÓN

En el capítulo anterior se han establecido los parámetros que definen la metodología que se ha propuesto para inyectar fallos, bien sean de tipo software o hardware, aunque principalmente como se ha descrito la intención es inyectar fallos de diseño del software, para evaluar componentes COTS en sistemas empotrados de tiempo real basados en microcontroladores que dispongan de puerto Nexus™.

En el capítulo anterior se ha definido el modelo de fallo que se va a inyectar, el modelo de sincronización entre la herramienta de inyección y el sistema a validar, la localización del fallo y los pasos a seguir para la obtención de las correspondientes medidas. Con el objetivo de validar dicha metodología se ha desarrollado una herramienta de inyección de fallos basada en el estándar Nexus™ [Pardo05b] que a continuación se va a describir.

Al desarrollar la herramienta con el objetivo de verificar y validar la metodología propuesta, aparece la disyuntiva sobre si desarrollar un dispositivo propio capaz de procesar y almacenar todos los mensajes que produce la interfaz Nexus™, o si de utilizar una herramienta de depuración comercial que nos permita implementar en un plazo de tiempo menor la metodología propuesta. Pues bien, se ha optado por la utilización de una herramienta comercial de las que hay actualmente disponibles en el mercado, que consiste en un sistema empotrado que implementan el protocolo de comunicaciones con la circuitería Nexus™ del microcontrolador. Dicho sistema está conectado al computador mediante un enlace USB o Ethernet.

A continuación se va a describir la herramienta y cada uno de los componentes que la integran, examinando las capacidades y características de los mismos. Posteriormente y una vez se estudie cómo funciona la herramienta, se van a describir las diferentes cargas que se han utilizado para la realización de los experimentos cuyos resultados se pueden encontrar en el capítulo siguiente.

6.2 HERRAMIENTA DESARROLLADA

La herramienta se ha desarrollado con la finalidad de poder realizar una inyección de fallos software y hardware, que no introduzca ningún tipo de sobrecarga temporal ni espacial, sobre un sistema empotrado de tiempo real con puerto Nexus™. Para conectar la herramienta de inyección de fallos con el sistema empotrado se ha decidido utilizar un depurador Nexus™ comercial del fabricante Lauterbach [lauterbach02]. La idea es reproducir un entorno de desarrollo real donde se pueda verificar y validar componentes COTS funcionando en SoC's.

Como modelo de fallo se ha implementado la inversión o "*bit-flip"*, y se pueden inyectar fallos de tipo inversión sencillos o múltiples. Para la sincronización de la inyección, el disparo que se ha implementado es del tipo temporal, es decir, la inyección se lleva a cabo transcurrido un tiempo tras el inicio del experimento, o espacial cuando la inyección se realiza al pasar por un punto determinado del código, todo ello en el caso de una inyección "*runtime"*. O como alternativa ésta puede realizarse antes de lanzar la aplicación a ejecución y entonces se habla de inyección "*pre-runtime"*, pudiendo posteriormente en tiempo de ejecución volver a cambiar el contenido de las posiciones de memoria que antes de la ejecución fueron sustituidas y alternar con una inyección de fallos *runtime*.

En última instancia cabe recordar que se pueden inyectar fallos tanto en memoria de programa como en memoria de datos. Aunque como se ha descrito en el capítulo anterior, el objetivo es inyectar sobre las zonas de memoria en las que se lleva a cabo el pase de parámetros de las llamadas a función de los diferentes componentes software que integran el sistema. Así dicha inyección se llevará a cabo en la zona de memoria de código, que como se vio en el capítulo anterior contiene las instrucciones donde se lleva a cabo la carga de los parámetros de las llamadas a función en los registros internos del microcontrolador.

A grandes rasgos se observa que la herramienta está básicamente constituida por tres módulos bien diferenciados:

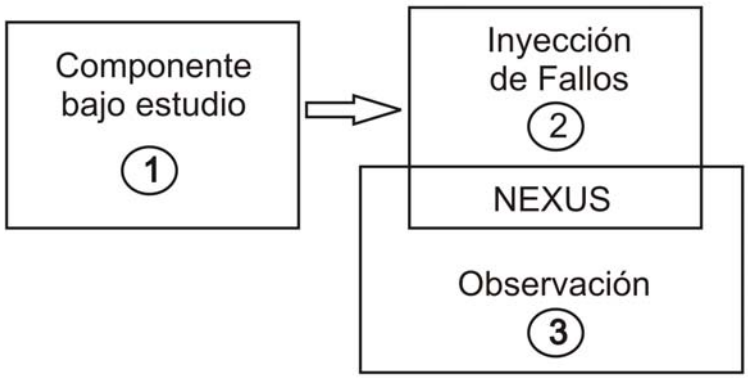


Figura 6.1: Módulos básicos de la arquitectura de la herramienta

El primer módulo al que se ha denominado como "*Componente bajo estudio*", se corresponde con la etapa donde se lleva a cabo un análisis previo de las direcciones de memoria donde se va a efectuar la inyección de fallos. En el segundo módulo, y una vez que se han seleccionado dichas direcciones de memoria, se pasa a la etapa donde se va a proceder con la propia inyección de fallos utilizando las capacidades que el puerto Nexus™ ofrece, esta fase se ha denominado como "*Inyección de fallos*". Y en último lugar, una vez se han realizado los experimentos en inyección de fallos, se pasa a la última fase o módulo que se ha denominado como de "*Observación"*, donde se van a tratar los resultados obtenidos de la experimentación para cada una de las cargas que se han testeado, utilizando también en este caso las facilidades

que el puerto Nexus™ provee para la monitorización del sistema. A continuación se va a describir con mayor detalle cada uno de los módulos que componen la herramienta.

6.3 MÓDULOS QUE COMPONEN LA HERRAMIENTA

En la siguiente figura 6.2, se puede observar de forma global cada uno de los módulos, y por tanto cada una de las etapas por las cuales la herramienta pasa para llevar a cabo las campañas de inyección de fallos.

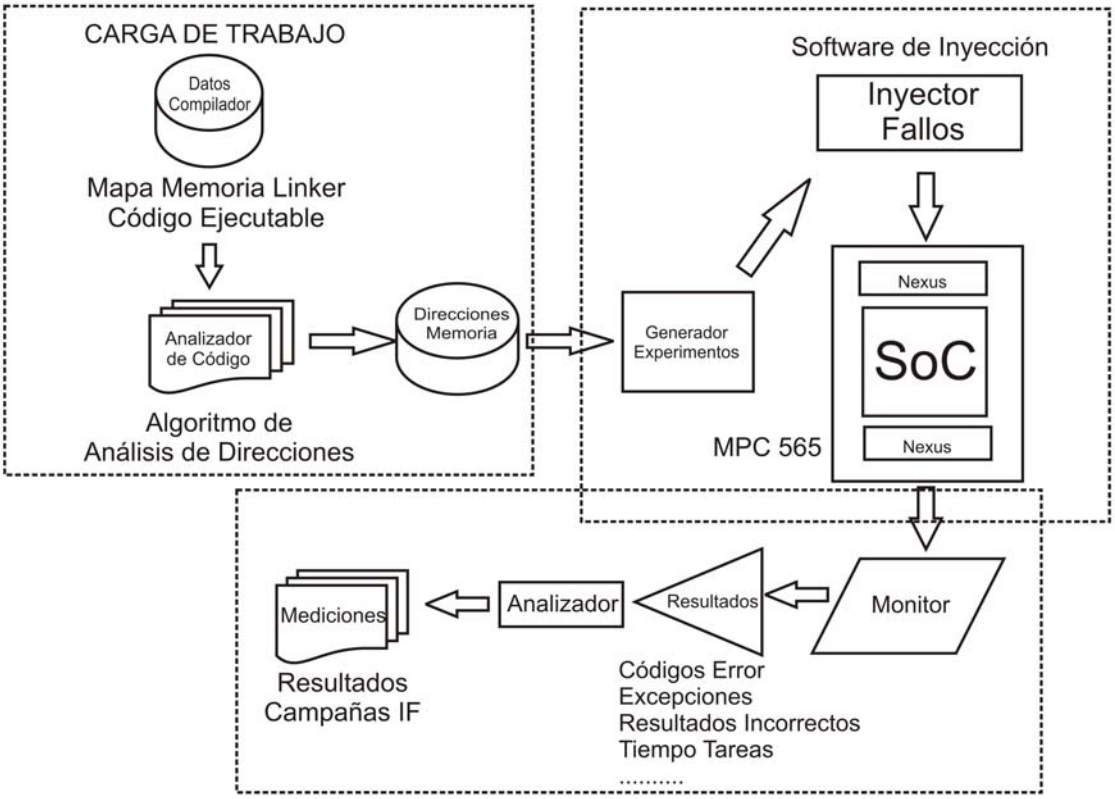


Figura 6.2: Detalle de los módulos y esquema general de la herramienta

Los recuadros punteados, que marcan cada uno de los módulos que anteriormente se han mencionado, esta vez muestran en su interior la arquitectura interna de cada uno de los componentes de la herramienta de inyección de fallos desarrollada. Como se puede observar en la parte superior izquierda de la figura 6.2 se parte de una carga de trabajo, constituida por diferentes componentes COTS, sobre la cual se quiere realizar una serie de experimentos de inyección de fallos. A partir de ahí se obtienen en primer lugar, las direcciones de memoria donde se va llevar a cabo la inyección de fallos y toda una serie de parámetros que se necesitan para configurar los diferentes experimentos, y que posteriormente serán ejecutados en el siguiente módulo correspondiente al propio inyector de fallos. Dicho componente es el encargado de realizar una serie de experimentos sobre la carga que se ejecuta en el microcontrolador. Finalmente y una vez se ha monitorizado el comportamiento del sistema, se filtran y analizan las diferentes medidas obtenidas para acabar presentando los resultados de las diferentes campañas en inyección de fallos que se han efectuado.

A continuación se va a describir con más detalle cada uno de los pasos que se siguen dentro de cada uno de los módulos que definirían la herramienta que se ha desarrollado.

6.3.1 Modulo de generación de los experimentos

En primer lugar se va a describir el módulo que se ha denominado como “Generador de experimentos”. Como muestra la figura 6.3 se parte de una carga de trabajo constituida por un software construido a partir de la integración de diferentes componentes COTS. En este caso como ejemplo se tiene que la carga de trabajo va a estar constituida básicamente por un sistema operativo de tiempo real y una aplicación que funciona por encima haciendo uso de los servicios que éste le proporciona.

Como se ha comentado en sucesivas ocasiones, el principal objetivo es inyectar fallos de diseño del software con la finalidad de poder evaluar la robustez de un componente determinado. Es por ello que el propósito de la inyección será actuar sobre el medio de comunicación con el exterior que el componente tiene para interactuar con otros componentes, considerando que todo componente tiene una interfaz pública que centraliza la recepción de toda la información que le llega desde el exterior. Considerando por tanto que dicha interfaz es el único conducto a través del cual los componentes interactúan con su entorno [Pardo06a]. Así que la meta será realizar la inyección de fallos en la API de los componentes, considerándolos a éstos como cajas negras donde se supone que no se tiene acceso al código fuente de los mismos.

Dicha interfaz está constituida por una serie de funciones que contienen un conjunto de parámetros de entrada y salida. Pero como se ha explicado en el capítulo anterior, debido a consideraciones de rendimiento y con el objetivo de acelerar la ejecución de las aplicaciones, la mayoría de los compiladores utilizan los registros internos del sistema para llevar a cabo el intercambio de parámetros durante la invocación de dichas funciones. Y modificar el contenido de los registros internos supone tener que parar la ejecución del sistema. Por tanto, como se ha explicado en el capítulo anterior, si se quiere que la inyección de fallos no detenga la ejecución del sistema y por tanto sea no intrusiva, entonces se debe capturar el momento antes de que se lleve a cabo la carga de los parámetros en los registros internos del microcontrolador.

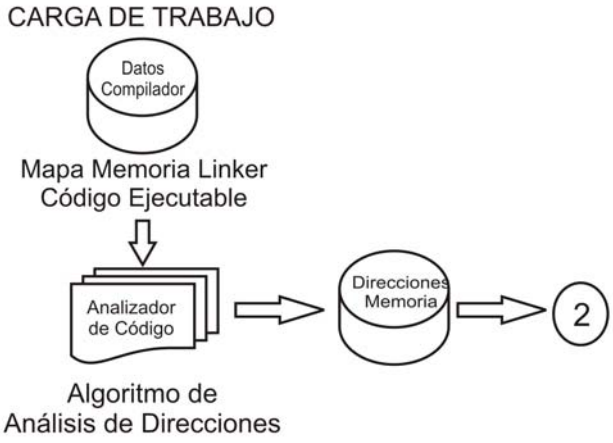


Figura 6.3: Módulo de generación de los experimentos

Por tanto a partir del mapa de memoria que proporciona el “*linker*” además del código ejecutable de la aplicación se va a obtener el conjunto de direcciones de memoria donde se lleva a cabo la carga de los parámetros de las llamadas en los registros internos del sistema, aunque como ya se ha comentado tan sólo bastaría con la información que el *linker* proporciona sobre el mapa de memoria. Para ello en primer lugar es necesario identificar el código máquina correspondiente a la zona de memoria donde se halla ubicada la aplicación, con esto se genera un fichero de texto. A partir de dicho fichero de texto y siguiendo el algoritmo que se ha planteado en el capítulo anterior, se lleva a cabo un análisis para obtener el conjunto de

direcciones sobre las cuales se va a realizar la inyección de fallos. Una vez realizado este análisis, el último paso consiste en generar el fichero definitivo de configuración de los experimentos, que contiene todos los parámetros de entrada que la herramienta necesita, de entre ellos las direcciones que se ha obtenido, para que el inyector empiece a generar las diferentes campañas de inyección de fallos.

6.3.2 Modulo de realización de los experimentos

Una vez se ha llevado a cabo el análisis de las direcciones de memoria donde realizar la inyección de fallos y teniendo todos los parámetros necesarios para la realización de los experimentos se empieza con la ejecución de las campañas de inyección de fallos.

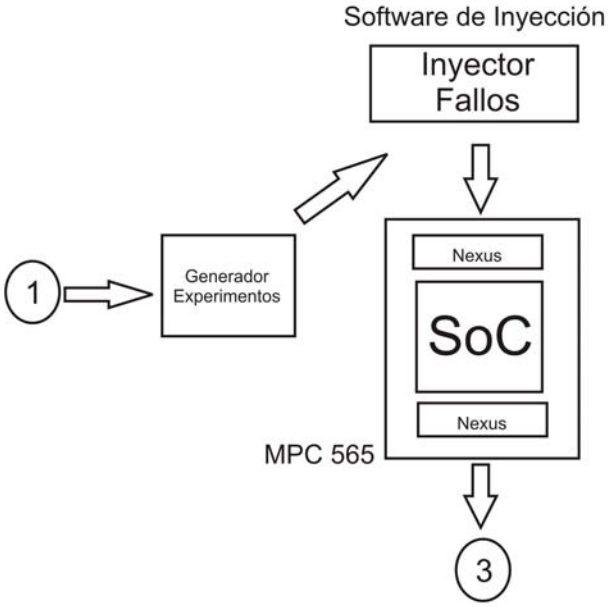


Figura 6.4: Módulo de realización de los experimentos

A partir de ese momento y antes de empezar con la ejecución de las campañas se decide si la inyección será efectuada durante la ejecución, en ese caso inyección "*runtime"*, o si se realizará antes de la ejecución, en ese caso inyección "*pre-runtime*". Si la inyección es *runtime* se tiene que configurar también el momento de la inyección. Ese momento se inicializa con el instante de tiempo en el cual comienza el experimento. Por tanto, una vez se tiene toda la información necesaria para realizar los experimentos, el inyector se pone en marcha, donde a través de la utilización de los mecanismos de escritura *on-the-fly*, que ofrece la interfaz Nexus™ para poder sustituir el contenido de zonas de memoria sin perturbar la ejecución del sistema, es decir sin tener que parar la ejecución, se van ejecutando los diferentes experimentos.

Para cada experimento el módulo inyector comienza con una ejecución donde no se lleva a cabo ninguna inyección de fallos y se observa al sistema en su funcionamiento normal, a esta ejecución se la denomina como "*golden-run*" [Koopman02]. Posteriormente se lleva a cabo una ejecución idéntica, bajo las mismas condiciones de entorno y ejecución, aunque en este caso sí que se inyecta un fallo. El objetivo de la ejecución libre de fallos (*golden-run*) es poder tener una referencia del funcionamiento del sistema para poder compararlo con la ejecución con inyección de fallos y así poder comparar ambas ejecuciones para ver cuál es el resultado del experimento.

Previamente a cada experimento se provoca un reset en el sistema y se carga la aplicación para tener un punto de partida uniforme entre experimentos. Se programa la traza para obtener las observaciones necesarias para el análisis, en caso de una inyección *runtime* se programa la secuencia de activación, sino la inyección se lleva a cabo en ese momento sustituyendo el contenido de la zona de memoria elegida. Posteriormente, se lanza la ejecución y se espera a que transcurra el tiempo que se ha establecido como tiempo total de ejecución de la aplicación. Tras la finalización de ésta se guardan las diferentes trazas en sus archivos correspondientes y se comienza con el experimento siguiente. Para la inyección del fallo se lee el contenido de la palabra de memoria sobre la que se va a inyectar, se modifica aplicándole la inversión correspondiente y se vuelve a escribir el valor resultante en la misma posición.

Durante la evolución de los experimentos, la herramienta de inyección abre una ventana donde se informa del número de experimento que se está realizando, del tipo de experimento del que se trata (libre de fallos o de inyección) y los resultados del mismo.

6.3.3 Modulo de análisis de los experimentos

Como anteriormente se ha descrito, el generador de experimentos permite definir de forma sencilla el conjunto de experiencias a realizar. Pero para posibilitar el análisis posterior, además de lanzar las ejecuciones con inyección de fallos, se hace necesario lanzar una serie de ejecuciones libres de de los mismos. El objetivo es poder evaluar las incidencias que los fallos introducidos provocan en la ejecución del sistema.

Por tanto, una vez se comienza con la ejecución de los experimentos se tienen que activar una serie de mecanismos que permitan observar cuál es la evolución del sistema, aunque dicho proceso de observación tiene que evitar cualquier perturbación o parada de la ejecución del mismo.

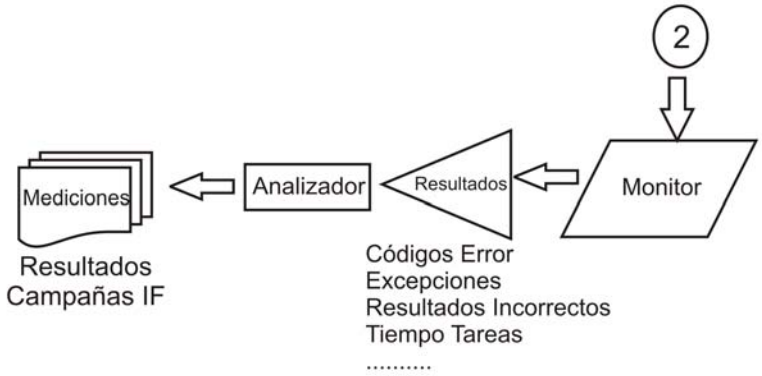


Figura 6.5: Módulo de análisis de los experimentos

Así, dentro de una secuencia de experimentación primero se lanza a ejecución la *golden-run* y se obtiene una traza de la ejecución libre de fallos. A continuación, se lanza la ejecución correspondiente al experimento con fallos donde también se almacenan sus correspondientes trazas.

Para la *golden-run* se establecerán una serie de puntos de observación o *watchpoints*, que permitan obtener en un momento determinado información sobre las variables de estado que proporcionan los valores de salida de la aplicación. Por otro lado, para la ejecución en la cual se lleva a cabo la inyección de fallos, además de establecer dichos puntos de observación, que nos permitirán poder comparar la finalización de ambas ejecuciones, además se tendrán que

establecer puntos adicionales de observación para poder monitorizar si el componente que está bajo observación es capaz de detectar errores en el sistema.

Con respecto a la información referente a los tiempos de ejecución de las tareas, tanto para la *golden-run* como para la ejecución con fallos inyectados, es también necesario establecer los correspondientes puntos de observación y así poder monitorizar los efectos de los fallos inyectados en el sistema en los tiempos de ejecución. De gran interés será poder observar si los *deadlines* establecidos para cada una de las tareas al final se cumplen o no a causa de los fallos introducidos.

6.3.4 Procedimiento experimental

Para cada uno de los experimentos se sigue el procedimiento que se describe continuación:

En primer lugar, se provoca un reset en el sistema, esto permite tener un punto inicial de partida común entre experimentos, posteriormente se inicializa el sistema para obtener el mismo marco de trabajo. A continuación se lanza una ejecución libre de errores (*golden-run*) y luego el experimento donde se inyectan fallos. Si la inyección es *pre-runtime* el fallo se inyecta en la fase de inicialización y se lanza la ejecución. Si la inyección es *runtime* se lanza la ejecución y se espera a la sincronización de la inyección del fallo.

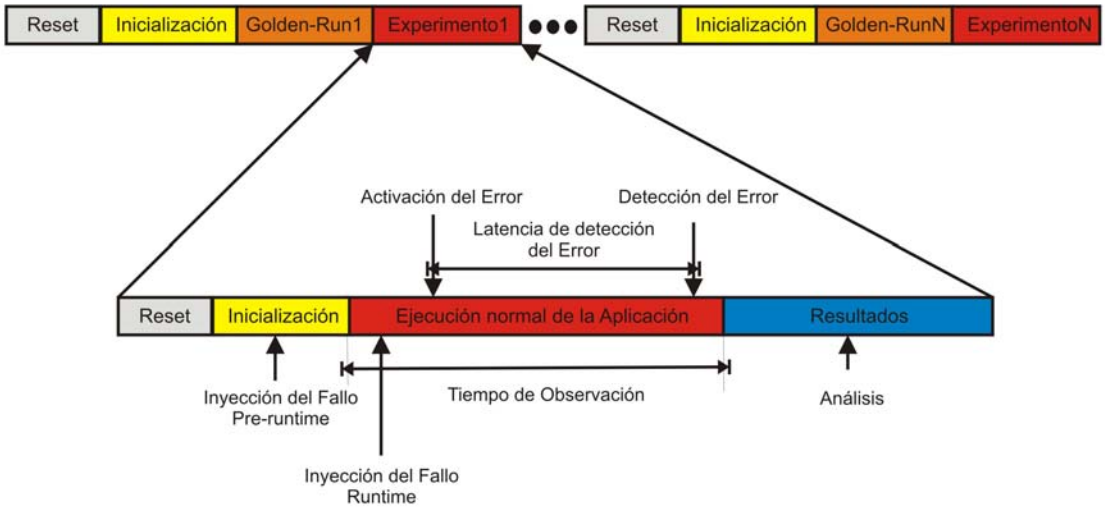


Figura 6.6: Procedimiento experimental

Una vez se ha inyectado el fallo, se considera que éste está latente hasta que el programa ejecutándose en el sistema a validar lea la posición de memoria que lo contiene. En ese momento se dice que el fallo se ha activado. Si el fallo produce un error y los mecanismos de detección de errores se activan, entonces se mide la latencia de detección del error, que será la diferencia entre el instante de activación del fallo y el instante de detección del error.

Tras inyectar el fallo, se observa la actividad del sistema a validar durante un periodo de tiempo que se ha llamado como "*tiempo de observación*". Este tiempo debe ser mucho mayor que el tiempo necesario para ejecutar un ciclo completo de procesamiento de la aplicación que se está ejecutando en el sistema a validar. En caso contrario, se podrían confundir experimentos en los que el fallo no se ha manifestado por falta de tiempo, con fallos que no afectan al sistema (aquellos que quedarían cubiertos por la redundancia intrínseca del sistema).

Al finalizar dicho tiempo de observación, para cada uno los experimentos se graban las trazas correspondientes a los valores de salida de la aplicación, los tiempos de las latencias de detección de errores y tiempos de ejecución de cada una de las tareas que se ejecutan en el sistema. Posteriormente se pasa a la fase de análisis de los resultados.

Una vez se ha recogida toda la información necesaria para la obtención de los resultados, en primer lugar se mira si el tiempo de observación se ha consumido o por el contrario se ha producido alguna excepción que haya detenido la ejecución del sistema. En tal caso se dice que los mecanismos de detección de errores del microcontrolador han actuado.

Si la ejecución finaliza tras ese tiempo de observación entonces se mira si los valores de salida de las variables de estado para ambas ejecuciones, libre de fallos y con fallos inyectados, presentan diferencias. En caso afirmativo se dice que la inyección de fallos ha producido un error en el sistema haciendo que su ejecución sea incorrecta, y por tanto se puede hablar de que al final se ha producido una avería del sistema. También se puede dar el caso de que la ejecución finalice tras el tiempo de observación establecido, pero la monitorización de la variable de estado indica que a partir de un momento determinado dicha variable no ha cambiado de estado, en este caso se habla de un cuelgue (situación de no respuesta) en el sistema y por tanto también de una avería del mismo. Así pues, en caso de avería, además se observará si alguno de los mecanismos de detección de errores de alguno de los componentes ha emitido alguna notificación a modo de código de error, en caso afirmativo se dirá que la avería ha sido cubierta, y por tanto se habla de una avería asegura. Si finalmente se ha producido una ejecución incorrecta y ninguno de los componentes del sistema ha detectado el error entonces se habla de averías no seguras. Por último, si tras la inyección de fallos el sistema funciona correctamente, es decir al final no se producen averías, entonces se dice que los mecanismos de tolerancia a fallos intrínsecos del sistema han tolerado dicho fallo.

Con respecto a los tiempos de ejecución de las tareas en primer lugar, se deben llevar a cabo una cantidad representativa de experimentos libres de fallos para obtener datos estadísticos sobre los tiempos de ejecución de las tareas en su funcionamiento normal. Esto siempre y cuando no tengamos la referencia de tiempos, que el diseñador de una aplicación en tiempo real pudiera tener establecidos como requisitos. Si los *deadlines* de las tareas ya son conocidos, en este caso tras la realización del experimento y observando la tasa de ejecución de los tiempos máximos que han llegado a alcanzar dichas tareas, se puede observar si se ha producido algún tipo de avería desde el punto de vista temporal (en el domino del tiempo). Observando, en este caso, averías por retraso en la entrega del servicio, cuando los valores que se obtienen por parte de la aplicación se producen más tarde de lo esperado, o averías por adelanto cuando los resultados que se esperan obtener se procesan antes del tiempo requerido.

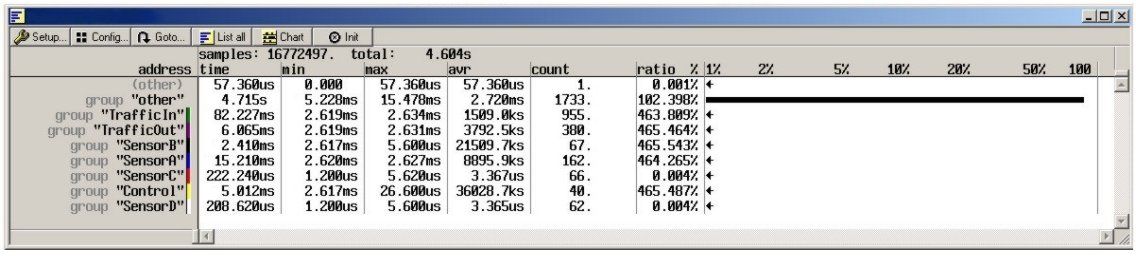


Figura 6.7: Tiempos de ejecución de las diferentes tareas del sistema

En la figura 6.7 se puede observar un ejemplo de traza de los datos que se han obtenido, en un experimento donde se ha inyectado un fallo, para calcular los tiempos mínimos, máximos, promedios y las veces que una tarea determinada pasa a ejecutarse durante un tiempo total de observación, que en este caso como se puede ver en dicha figura 6.7 ha sido de 4,604s. Los colores asignados a cada una de las tareas indican los diferentes grupos que se han establecido

de acuerdo a la metodología que se ha propuesto en el capítulo anterior y que distingue a las diferentes tareas que se están ejecutando en el sistema.

6.4 ENTORNO DE EXPERIMENTACIÓN

A continuación se describe cuáles han sido las herramientas que constituyen el entorno de experimentación que sea utilizado para llevar a cabo los experimentos de inyección de fallos. En primer lugar, para ejecutar la carga que se va a evaluar se ha utilizado como procesador el microcontrolador PowerPC MPC565 de Motorola. Microcontrolador de 32bits que funciona a 40MHz y que implementa la especificación clase-3 de la interfaz Nexus™. Como placa de evaluación de este procesador, se ha utilizado la CMD-565 del fabricante Axiom Manufacturing, que incluye el procesador y la circuitería de alimentación necesaria junto con el conjunto de conectores que facilitan el acceso a todos los pines del microcontrolador, como es el caso de los conectores correspondientes al puerto Nexus™ del que dispone este microcontrolador.

Por otro lado, para la monitorización/depuración de dicho puerto Nexus™ se ha utilizado el depurador comercial TRACE32-PowerTrace/NEXUS™ de la casa alemana Lauterbach GMBH. Dicho depurador provee de 96 canales de traza pudiendo llegar a funcionar éstos hasta los 200MHz y con una profundidad de traza de 16MFrames. Además cada traza viene acompañada con una marca temporal de 32 bits y una resolución de 20ns. En su modo normal de funcionamiento este depurador se conecta por un extremo al sistema en desarrollo mediante el conector Nexus™. Por el otro extremo, utiliza un puerto USB de un computador tipo PC-compatible y mediante un entorno gráfico propio permite realizar las funciones típicas de un depurador comercial, como cargar programas en memoria, ejecutarlos o parar la ejecución. Para este trabajo se ha aprovechado la posibilidad que ofrecen estos depuradores de programar su funcionamiento mediante un lenguaje propietario, en este caso llamado *"Practice"*, programado así las funciones de la herramienta de inyección de fallos en este lenguaje. Por último, el computador utilizado para dirigir las campañas de inyección de fallos es un PC convencional.

En la siguiente figura 6.8 se puede ver el depurador Nexus™ conectado a la placa de evaluación.

Figura 6.8: Entorno de experimentación

La herramienta se ha implementado básicamente como un *"script"* o archivo de órdenes escrito, en dicho lenguaje Practice, que implementa cada uno de los módulos que la componen. La herramienta Trace32 de Lauterbach utiliza este lenguaje para permitir la programación de

secuencias complejas de ejecución de las órdenes del depurador y da la posibilidad de definir las diferentes ventanas y contenido de las mismas, que constituyen la interfaz de usuario de la herramienta que se desarrolle. En la figura 6.9 se puede ver un ejemplo de la interfaz gráfica correspondiente a las campañas de inyección de fallos de una de las cargas que se ha utilizado. En la esquina inferior derecha de la figura se puede observar los diferentes mensajes que la herramienta ofrece al usuario a medida que los experimentos se van ejecutando. En el resto de las ventanas se pueden observar datos relativos a la ejecución de las tareas, alarmas y eventos que se van sucediendo durante la ejecución, así como el código máquina y en alto nivel que se va ejecutando.

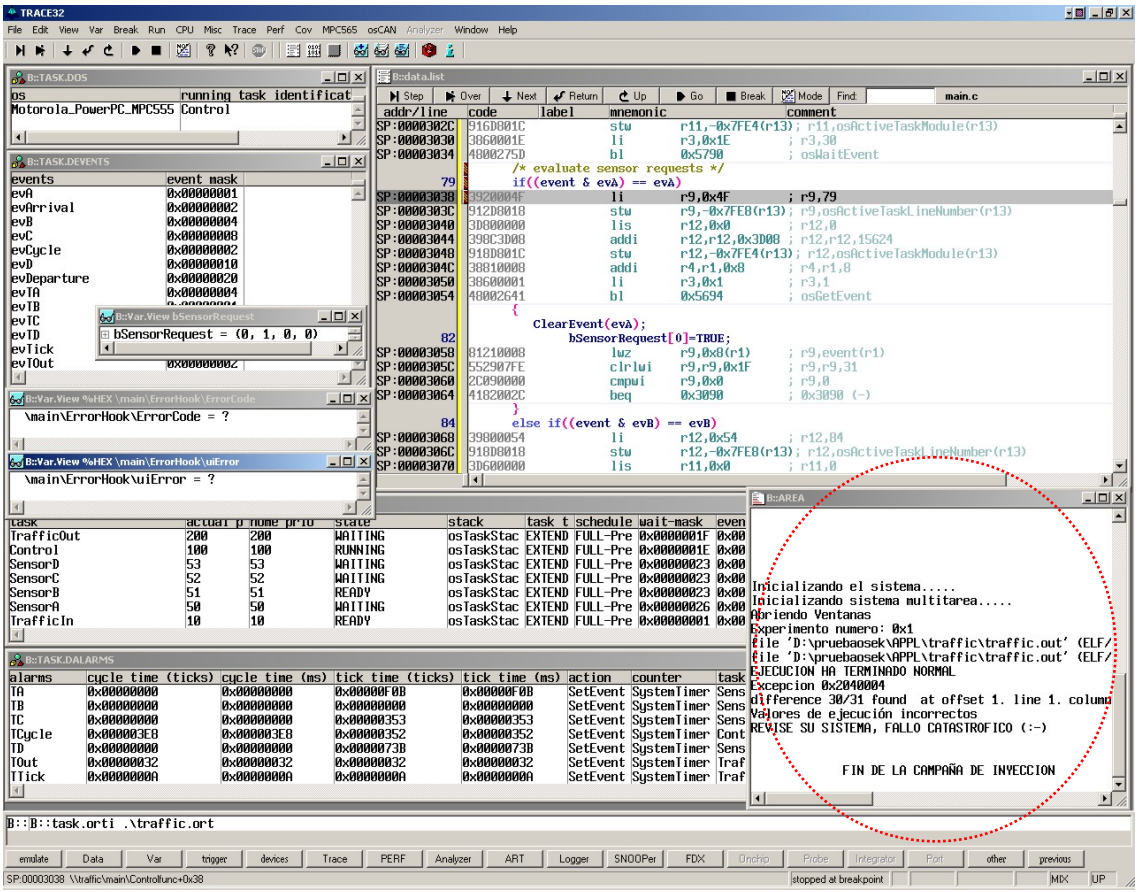


Figura 6.9: Ejecución del "*Script*" de la herramienta de inyección de fallos.

6.5 DESCRIPCIÓN DE LA CARGA DE TRABAJO

En este apartado se va a describir cuáles han sido las diferentes cargas de trabajo que se han utilizado para validar la metodología que se propuso en el capítulo anterior. Como posteriormente se verá en el capítulo de los resultados, se han realizados experimentos con tres cargas diferentes. En primer lugar la carga que se ha probado estaba constituida por el sistema operativo MicroC/OS-II y una aplicación que simula el control electrónico de un motor diésel [DBench04]. En segundo lugar y con la finalidad de poder realizar una comparativa, se ha utilizado una misma aplicación de un control de semáforos funcionando sobre dos RTOS distintos como son MicroC/OS-II y una implementación de la especificación OSEK/VDX.

6.5.1 Descripción de la aplicación de un control de motor diésel

Para la primera carga se ha utilizado una aplicación que implementa una versión reducida de una ECU (del inglés "*Electronic Control Unit*") de motor diésel basada en el microcontrolador MPC565 de Motorola que incluye el estándar Nexus™. Se trata de un sistema de control de un motor diésel con inyección del tipo "*common-rail*". Esta tecnología de inyección de combustible se caracteriza por utilizar una bomba para presurizar el mismo raíl dentro de un depósito de presión. Los inyectores de combustible están conectados al raíl y se abren a intervalos regulares para introducir la cantidad precisa de combustible dentro de los cilindros. Tanto la presión en el raíl como la temporización de las inyecciones se deben regular con precisión. Esta tecnología se ha podido popularizar gracias a la utilización de sistemas electrónicos de control basados en microprocesador. Sin embargo los sistemas que la implementan deben tener la confiabilidad necesaria para garantizar el funcionamiento correcto, ya que una avería del sistema de control que haga que genere resultados incorrectos podría provocar daños irreversibles en la mecánica del motor e incluso poner en peligro la integridad de los ocupantes del vehículo.

En este trabajo se ha utilizado una versión reducida [Yuste03c] que se utilizó en el proyecto DBench ("*Dependabilily Benchmarking*") para desarrollar un *benchmark* de confiabilidad para aplicaciones empotradas de control de motores de combustión interna. Se trata de una versión reducida ya que se utiliza un menor número de sensores, actuadores y tablas de parámetros. Se considera que al reducir la aplicación no se ha reducido también su representatividad, ya que los sensores, actuadores y tablas eliminados sólo se utilizan en la optimización del funcionamiento del motor, no en su funcionamiento básico. Además se ha tenido la precaución de mantener una muestra de todos los algoritmos básicos de control que se utilizan en una ECU real. Estos algoritmos son reguladores PI (del inglés "*proporcional integral*") y reguladores todo/nada configurados mediante tablas de parámetros. De la misma manera se utiliza un control basado en búsqueda en tablas e interpolación de los resultados característico de las aplicaciones actuales de automoción.

Un motor de combustión interna es una máquina que quema combustible mezclado con aire y convierte en movimiento parte de la energía que se desprende. Para optimizar este proceso, se deben mezclar estos dos componentes en una proporción concreta. Por lo tanto nos encontramos con dos procesos diferentes, aunque acoplados, que deben ser controlados. Por un lado se tiene la inyección de combustible y por el otro la gestión de aire. En la figura 6.10 se puede ver en un esquema los diferentes bucles de control encargados de la supervisión y regulación de estos procesos que se han implementado como tareas del sistema.

El proceso de inyección está controlado por dos bucles de control. El primero es el bucle de control de temporización de la inyección, el segundo el bucle de control de la presión del raíl, o depósito de presión. El bucle de control de temporización de la inyección se basa en un algoritmo de búsqueda en tablas e interpolación. Estas tablas, también llamadas cartografía, contienen valores normalmente obtenidos de manera experimental o que representan los resultados de algoritmos complejos. Este procedimiento de buscar el resultado del algoritmo en una tabla se utiliza para reducir el tiempo de cálculo y por lo tanto la potencia necesaria para resolver el problema de control. En este caso se usan como entrada al algoritmo el régimen de giro del motor y la posición del acelerador. De las tablas se obtienen, a partir de estas entradas, los datos de ángulo de inicio y duración de las inyecciones de combustible y el dato de presión de consigna del raíl.

El bucle de control de la presión del raíl es un regulador PI (proporcional integral). Este regulador se encarga de mantener la presión real del raíl lo más cercana posible a la presión de consigna. La presión real del raíl es un dato medido mediante un sensor y fluctúa con el estado de funcionamiento del motor (régimen, consumo, etc.). De este parámetro, junto con el de

tiempo de apertura del inyector, dependerá la cantidad total de combustible que se inyecta en el motor. La salida del regulador que controla la presión del raíl es un valor en tanto por ciento, que se aplica al actuador del raíl mediante una señal modulada por anchura de pulso o PWM (del inglés "*pulse width modulation*").

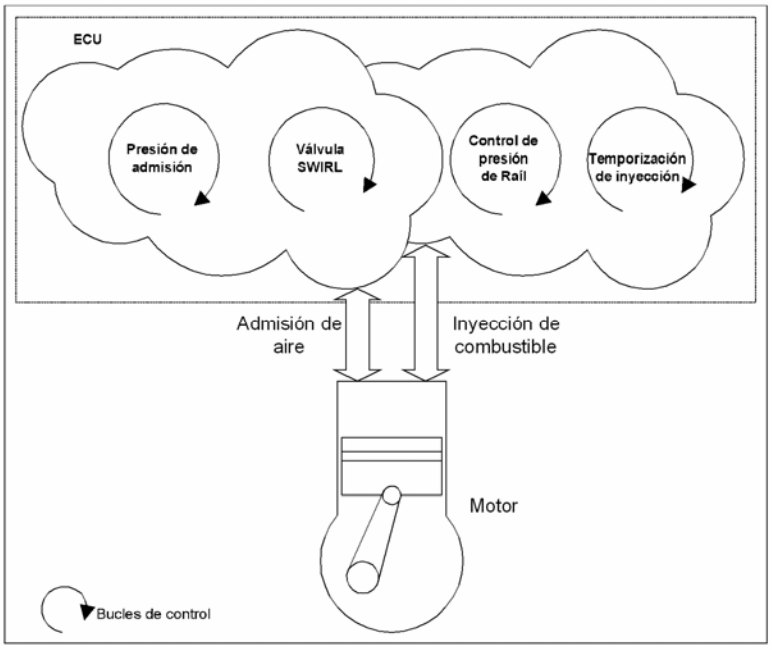


Figura 6.10: Aplicación de control de motor diésel

Para los bucles de control de la gestión de aire los reguladores que se han elegido son del tipo todo/nada. También hay dos bucles de control, el primero se encarga de regular la presión de admisión y el segundo de actuar sobre la válvula SWIRL. Del primer parámetro (presión de admisión) depende directamente la cantidad de aire disponible en el motor. Un turbo compresor utiliza la energía de los gases de escape para aumentar esta presión. La regulación se hace mediante una electroválvula llamada "*Waste-gate*" que al abrirse elimina presión de la turbina reduciendo la presión de admisión. Sobre esta válvula se hace una regulación del tipo todo/nada para mantener la presión de admisión cerca de su consigna. Esta consigna, como en el caso anterior, es extraída de las tablas para el punto de funcionamiento actual del motor.

El segundo bucle de control de aire actúa sobre una válvula, llamada de SWIRL, que modifica la geometría de admisión. Esta válvula tiene dos estados. En el primer estado el aire entra al colector de admisión del motor por un sólo conducto que tiene una forma especial para crear turbulencias en el aire. En el segundo estado se abre un segundo conducto de forma que no se crean tantas turbulencias pero aumenta el caudal de aire disponible. Como en los casos anteriores, de las tablas se obtiene en cuál de estos estados debe estar la válvula.

6.5.2 Descripción de la aplicación de un Control de Semáforos

En segundo lugar se ha utilizado una aplicación de control de semáforos que se ha probado sobre dos sistemas operativos de tiempo real diferentes. El objetivo es ver si la utilización de uno u otro RTOS, aporta diferencias respecto a la confiabilidad y robustez final del sistema. Dicha aplicación de control, que es menos compleja en su desarrollo que la anterior, simula el tráfico en la intersección de dos calles con vehículos circulando en ambos sentidos. El sistema consta de cuatro semáforos y una unidad de control para todos ellos. Además cada semáforo

cuenta con un sensor que monitoriza el tráfico de vehículos. En la siguiente figura 6.11 se puede ver un esquema de la situación de los semáforos y los sensores:

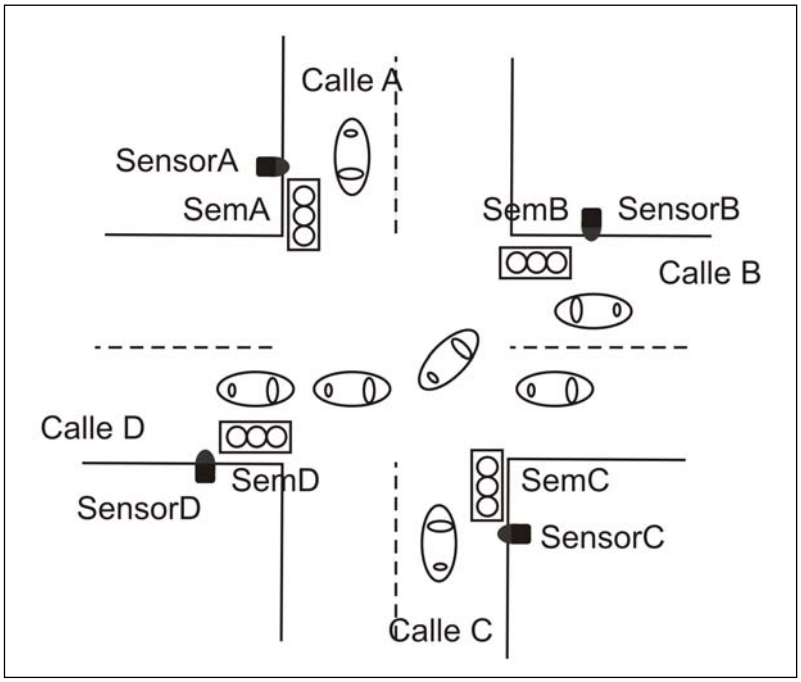


Figura 6.11: Aplicación de control de semáforos

El sistema consta de las siguientes tareas que simulan todo el proceso llevado a cabo para el control de los semáforos en el cruce:

- Tarea "***TrafficIn***": esta tarea simula el tráfico de llegada a cada calle. Para evitar una llegada de vehículos de manera aleatoria, se ha implementado una tasa equidistante de llegada de vehículos, donde cada cierto periodo de tiempo llega un vehículo al semáforo de una calle determinada.

- Tareas "***Sensor***": hay cuatro tareas "*Sensor*" correspondientes a cada uno de los cuatro semáforos que existen. Estas tareas monitorizan de tráfico de llegada y salida en cada calle. En este caso se tiene que si hay "*n*" vehículos esperando en el semáforo o hay al menos un vehículo esperando un tiempo "*t*" determinado, dicho sensor envía una señal a la tarea "*Control*" para notificar tal evento.

- Tarea "***Control***": esta tarea se encarga de controlar los eventos que se suceden en cada una de las calles siguiendo una estrategia determinada donde se atiende en un bucle sin fin a cada una de las calles un periodo de tiempo determinado. En este caso se tiene que si para una calle determinada, no se recibe ninguna notificación por parte del sensor correspondiente, ésta es saltada y se pasa a monitorizar la siguiente. El objetivo es ver que semáforo debe ser cambiado. Tras controlar que semáforo debe ser cambiado la tarea notifica este suceso a la tarea "*TrafficOut*".

- Tarea "***TrafficOut***"; finalmente esta tarea simula el tráfico de salida de cada calle, cambiando el estado de las luces del semáforo correspondiente. Dando en este caso paso a los vehículos que han estado esperando a que alguno de los dos eventos anteriormente comentados se haya cumplido.

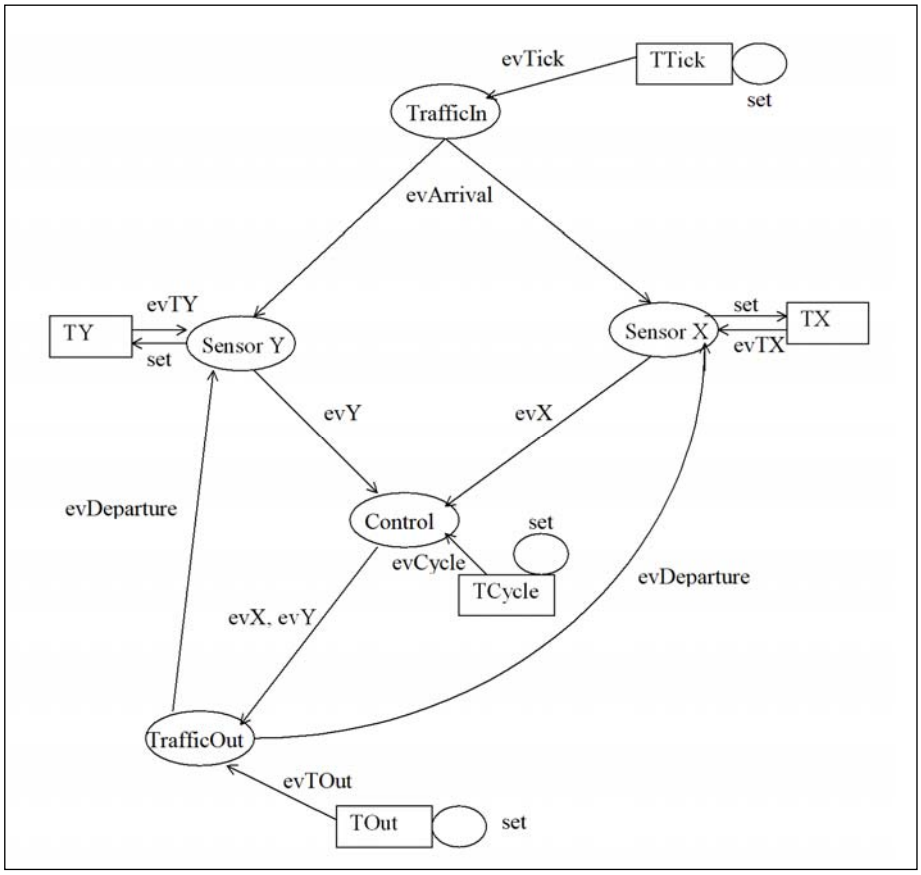


Figura 6.12: Topología para dos semáforos.

En la figura 6.12 se puede observar cómo se implementaría la sucesión de eventos para el caso de dos semáforos, aunque por simetría la misma solución ha sido aplicada para los cuatro. En el diagrama se puede ver como se conectan las diferentes tareas que se ejecutan de forma infinita en el sistema. Los eventos que cada una espera de las otras y un evento de tipo temporal que marca el tiempo máximo que la tarea estará en espera en caso de que los eventos que espera no lleguen.

6.6 SISTEMAS OPERATIVOS DE TIEMPO REAL (RTOS)

A continuación se va a realizar una pequeña descripción de cada uno de los sistemas operativos de tiempo real (RTOS) que se han utilizado en la realización de los experimentos, como ejemplo de componentes COTS que podrían ser incorporados a un desarrollo de software.

Se ha utilizado dos sistemas operativos de tiempo real como son MicroC/OS-II y una implementación de OSEK/VDX. Decir que si bien MicroC/OS-II se ha probado con ambas aplicaciones, la que simulaba la implementación de un control electrónico de un motor diésel y la del control de semáforos en una intersección, para el caso de la implementación de OSEK/VDX sólo se ha probado la aplicación del control de semáforos. El objetivo era ver si con otro sistema operativo y otro compilador se podía verificar y validar la metodología de inyección de fallos que se ha propuesto en el capítulo anterior.

6.6.1 MicroC/OS-II

MicroC/OS-II del inglés "*MicroController Operating System*", es un sistema operativo de tiempo real utilizado en muchos sistemas desde hace ya algunos años, ya con su primera versión MicroC/OS hacia 1992. Desde entonces se puede encontrar aplicaciones y sistemas como cámaras, robots industriales, control de motores, instrumentos médicos, etc., en los que se ha utilizado este sistema operativo. Actualmente la versión más reciente es la V2.52 que ha sido certificada para sistemas críticos de aviación, cumpliendo las normas RTCA DO-178B y EUROCAE ED-12B, para dispositivos médicos la FDA 510(k), y la IEC-61058 para sistemas nucleares y de transporte. Además y según anuncian en su página web[4] MicroC/OS-II ahora es 99% compatible con los estándares de codificación de C establecidos por MISRA (del inglés "*Motor Industry Software Reliability Association*") para sistemas críticos dentro del mundo de la automoción [Validated02]. Dicho consorcio incluye a Delco Electronics, Ford Motor Company, Jaguar Cars Ltd., Lotus Engineering, Lucas Electronics, Rolls-Royce, Rover Group Ltd., etc. donde uno de sus objetivos es mejorar la fiabilidad y seguridad de los dispositivos electrónicos del automóvil.

A modo de resumen, como aspectos genéricos de este RTOS se pueden destacar las siguientes características [Labrosse01]:

- Portable: MicroC/OS-II está escrito en ANSI C, con el código específico para cada microprocesador en lenguaje ensamblador, facilitando así más aún su portabilidad. Además funciona para la mayoría de los microprocesadores, microcontroladores y DSPs (del inglés "*Digital Signal Processor")* de 8, 16, 32 y 64 bits.

- Empotrable: Ha sido diseñado para sistemas empotrados. Lo cual permite que sea albergado en dispositivos donde el espacio de memoria es limitado.

- Escalable: MicroC/OS-II está diseñado para que según la aplicación sólo se utilicen aquellos servicios que se requieren, haciendo que la cantidad de memoria necesaria sea menor. La escalabilidad se consigue a través de la compilación condicional. Esto reduce la cantidad de espacio requerido tanto para datos y como código.

- Expulsivo: El núcleo es de planificación expulsiva en tiempo real, significando esto que siempre se ejecuta la tarea de mayor prioridad lista para ejecución.

- Multitarea: Se puede manejar hasta 64 tareas, sin embargo el sistema se guarda 8 para su uso propio, lo cual nos deja con 56 tareas para la aplicación de usuario. Cada tarea tiene una única prioridad asignada que la identifica.

- Determinista: El tiempo de ejecución de todas las funciones y servicios es totalmente determinista. Lo cual significa que siempre se puede saber cuanto se tarda en ejecutar una determinada función o servicio.

- Dotado de Pila para las Tareas: Cada tarea requiere de su propia pila, sin embargo se permite que cada tarea tenga una pila con un tamaño diferente según las necesidades de la aplicación, reduciendo así la cantidad de memoria necesaria. Con la característica de chequeado de pila que ofrece MicroC/OS-II, se puede saber exactamente cuanto espacio de pila requiere cada tarea.

[4] http://www.ucos-ii.com/index.html. Es esta web el lector podrá encontrar toda la información al respecto.

- Servicios: es un SO que provee de diferentes servicios tal como buzones, semáforos, colas, particiones de memoria de tamaño fijo, funciones de temporización y muchas más.

- Manejo de Interrupciones: Se permite que las interrupciones puedan suspender la ejecución de una tarea. Si una tarea de mayor prioridad es despertada como resultado de una interrupción, la tarea de mayor prioridad pasará a ejecutarse tan pronto como el resto de interrupciones anidadas se completen. Se permite hasta 255 niveles de anidamiento para las interrupciones.

6.6.2 OSEK/VDX

En mayo de 1993 OSEK surge como un proyecto común de la industria alemana del automóvil con el objetivo de desarrollar un nuevo estándar para definir una arquitectura abierta para unidades de control distribuidas en vehículos. OSEK es una abreviatura para el término alemán "und de Offene Systeme deren el dado Elektronik im Kraftfahrzeug del für de Schnittstellen" (del inglés "*Open Systems and the Corresponding Interfaces for Automotive Electronics*"). Los socios iniciales del proyecto eran BMW, Bosch, DaimlerChrysler, Opel, Siemens, VW y la Universidad de Karlsruhe como coordinador. Posteriormente los fabricantes franceses de vehículos como PSA y Renault se unieron al consorcio en 1994 aportando las siglas VDX (del inglés "*Vehicle Distributed eXecutive*"), proyecto similar dentro de la industria del automóvil francesa. Finalmente en octubre de 1997 aparece publicada la especificación OSEK/VDX [OSEK05]. Actualmente algunas partes de OSEK están siendo desarrolladas como estándares por ISO en la norma ISO 17356.

La arquitectura abierta introducida por OSEK/VDX abarca tres áreas principalmente[5]:

- Comunicación (para intercambio de datos en y entre unidades de control)
- Sistema operativo (para permitir la ejecución en tiempo real del software empotrado en una ECU y base para otros módulos de OSEK/VDX)
- Manejo de red (para tareas de configuración y supervisión)

La gran motivación del desarrollo de este estándar era la de reducir los elevados costos que se dan en el desarrollo de las diferentes variantes de aquellos aspectos no relacionados con el software de las unidades de control (ECUs) y superar la incompatibilidad de los diversos interfaces y protocolos implementados en unidades de control realizadas por diferentes fabricantes.

El objetivo era introducir una mayor portabilidad y reutilización del software creando una especificación de interfaces abstractos e independientes de las aplicaciones, y especificación de interfaces de usuario independientes del hardware y de la red. Además se planteaba la necesidad de tener un diseño eficiente de una arquitectura que permita que la funcionalidades sean configurables y escalables para un ajuste óptimo.

Las ventajas que se pretenden obtener con este nuevo estándar son:

- Considerables ahorros en costes y tiempos de desarrollo.
- Mejorar la calidad del software de las unidades de control de varios fabricantes.
- Establecer unas características de interconexión estandardizadas para las unidades de control con diversas arquitecturas.

[5] http://portal.osek-vdx.org/. Es esta web el lector podrá encontrar toda la información al respecto.

- Establecer una utilización ordenada de la inteligencia (recursos existentes) distribuida en el vehículo, y así mejorar el funcionamiento del sistema total sin requerir de hardware adicional.
- Proporcionar una independencia absoluta en lo que respecta a la puesta en práctica individual de cada fabricante, pues la especificación no prescribe aspectos de la implementación.

En cuanto a la especificación del sistema operativo se tiene que la versión más actual se corresponde con la V2.2.3 desarrollada en Febrero de 2005. Dicha especificación define la interfaz estándar para un SO basado en sistemas de un sólo procesador en unidades de control empotradas distribuidas; ofreciendo la funcionalidad necesaria para sistemas de control conducidos por eventos. Aunque hay que decir que OSEK (a partir de ahora se mencionará así al SO) se desarrolla principalmente con el objetivo de hacer más eficiente la utilización de los recursos para aplicaciones software de unidades de control del automóvil.

Gran parte de las aplicaciones del automóvil tienen importantes requerimientos de tiempo real, por lo que el sistema operativo debe ser capaz de ofrecer la funcionalidad necesaria para el manejo de sistemas de control conducidos por eventos. Como el sistema operativo tiene como objetivo poder ser utilizado en cualquier unidad de control, éste debería soportar aplicaciones con fuertes restricciones temporales en una amplia variedad de dispositivos hardware. Además el sistema operativo debería ser capaz de ofrecer un alto grado de modularidad y flexibilidad en la configuración de requisitos para poder utilizar cualquier microprocesador y unidades de control complejas. Otros factores importantes para las aplicaciones del automóvil son la necesidad de una baja utilización de recursos, una alta escalabilidad y fiabilidad, y todo ello con un coste asociado reducido. Como el sistema operativo es el elemento base en la integración de los diferentes módulos de software desarrollados por diferentes fabricantes, además tiene que tener características de portabilidad y así ofrecer posibilidades de reutilización del software.

Éstas son algunas de las características que la filosofía de OSEK contempla en el desarrollo de la especificación de dicho sistema operativo. Otras características adicionales interesantes que se puede encontrar en la especificación son:

- Estandarización de interfaces: donde la interfaz utilizada entre las aplicaciones y el sistema operativo se define a través de la utilización de servicios del sistema. Dichos servicios del sistema comprenden el manejo de tareas, sincronización de eventos, manejo de interrupciones, alarmas y tratamiento de errores.

- Comprobación de errores: el sistema operativo ofrece dos niveles de comprobación de errores para las diferentes fases de desarrollo de las aplicaciones. Así, se permite la comprobación de errores en las primeras fases de testeo del software.

- Escalabilidad: el sistema operativo permite definir diferentes mecanismos de planificación y características de configuración que hacen que se pueda trabajar con un número de recursos del hardware muy reducidos.

- Portabilidad: la estandarización de la interfaz, en cuanto a llamadas al sistema, definición de tipos y constantes, y permite que se puedan transferir aplicaciones software entre diferentes unidades de control sin tener que realizar grandes cambios, soportando dicha portabilidad a nivel de código fuente.

6.7 RESUMEN Y CONCLUSIONES

En este capítulo se ha descrito la herramienta que se ha desarrollado a partir de la metodología explicada en el capítulo anterior. Esta herramienta se ha diseñado para inyectar fallos hardware y de diseño del software en sistemas empotrados de tiempo real con componentes COTS, sin introducir ninguna sobrecarga temporal ni espacial en los mismos. El modelo de fallo implementado es la inversión o *bit-flip* y se puede llevar a cabo una inyección tanto *pre-runtime* como *runtime*.

La herramienta está formada por tres módulos: el generador de experimentos, el inyector y el módulo de análisis. El generador de experimentos realiza un primer análisis para establecer los parámetros que definirán el proceso de inyección, posteriormente el inyector de fallos lleva a cabo los diferentes experimentos de inyección de fallos, y finalmente el módulo de análisis que se encarga de devolver los resultados de la experimentación, en cuanto a coberturas de detección de errores, latencias de detección, y de activación de errores y averías, y tiempos de ejecución de las tareas bajo el efecto de los fallos inyectados.

En este capítulo se ha desglosado cada uno de los módulos que componen la herramienta, describiéndolos en sucesivos pasos hasta el mínimo detalle, para comprender cómo funciona ésta, cuya misión final era la de validar la propuesta de metodología para inyectar fallos de diseño del software en componentes COTS que se ha visto en el capítulo anterior.

También al final del capítulo se han descrito el entorno de experimentación y las dos aplicaciones que se han utilizado, correspondientes a un control electrónico para un motor diésel y una aplicación de control de semáforos. Por último se ha hecho una pequeña descripción de características generales de los dos sistemas operativos de tiempo real que se han utilizado como son MicroC/OS-II y OSEK, como ejemplo de componentes COTS que podrían ser utilizados en el desarrollo de un sistema empotrado de tiempo real.

Capítulo 7

RESULTADOS DE LA EXPERIMENTACIÓN

7.1 INTRODUCCIÓN

Una vez se ha descrito en el capítulo anterior la herramienta de inyección de fallos que se ha desarrollado, y visto cómo funciona cada uno de los módulos que la componen, a continuación se va a ver cuales han sido los resultados obtenidos tras la realización de los experimentos. Como también se describió en el capítulo anterior, se han llevado a cabo experimentos con tres cargas diferentes con la finalidad de validar la metodología y técnica propuestas, y así ver los diferentes resultados que se pueden obtener.

En primer lugar se presentan los resultados obtenidos para una carga compuesta por un RTOS como es MicroC/OS-II y una aplicación que simula una unidad de control electrónica (ECU) de un motor diésel, descrita anteriormente. Esta carga ha sido utilizada como continuación del trabajo realizado en INERTE donde, utilizando esta misma carga, se inyectaban fallos de tipo hardware y se verificaba que la utilización del estándar Nexus^{TM} se hacía factible para la implementación de una técnica de inyección de fallos hardware no intrusiva. En este caso y yendo más lejos, utilizando también la interfaz Nexus^{TM} se ha propuesto una nueva metodología y técnica también no intrusiva para introducir fallos de diseño del software en partes concretas de los diferentes componentes COTS que integran el conjunto software de una aplicación.

Para la segunda y tercera carga se ha utilizado una aplicación de control de semáforos, también descrita en el capítulo anterior, pero con dos sistemas operativos de tiempo real diferentes como son OSEK/VDX y MicroC/OS-II. El objetivo en este caso es validar la metodología y técnica propuestas, y ver los diferentes resultados que se obtienen incorporando uno u otro RTOS en un sistema determinado para una misma aplicación.

Así pues, en los resultados que se van a mostrar a continuación se han obtenido en primer lugar datos relativos a las coberturas de detección del SO utilizado y del sistema completo. También se ha obtenido datos relativos a los errores detectados por parte del SO, su frecuencia y significación. Por parte del microcontrolador, se han analizado las excepciones lanzadas también con su frecuencia y su significado. Y con respecto a los tiempos, se ha obtenido información relativa a las latencias de detección del SO y los tiempos máximos de ejecución de

las tareas con y sin fallos, mostrando la distribución de dichos tiempos por tarea, para ver el efecto de los fallos introducidos en los tiempos de ejecución de las mismas.

El capítulo consta de tres grandes apartados, correspondientes a las tres cargas que se han evaluado. Así, los resultados de dicha evaluación se presentan siguiendo una misma estructura dando toda la información necesaria al lector para poder recurrir a cada una de ellas de forma independiente.

7.2 RESULTADOS PARA MICROC/OS-II Y LA ECU

En este apartado se van a presentar los resultados obtenidos para un sistema que incorpora una carga de prueba que implementa una versión reducida de una ECU de un motor diésel y un sistema operativo de tiempo real como es MicroC/OS-II. Se han llevado a cabo 2000 experimentos y estos son los resultados de la experiencia.

7.2.1 Coberturas

En primer lugar se va a mostrar los resultados que se han obtenido en cuanto a coberturas de detección tanto del propio sistema operativo bajo estudio, en este caso MicroC/OS-II, como sobre la cobertura final del sistema. Para ello habrá que obtener, además de los datos correspondientes al SO, datos relativos al propio microcontrolador MPC565 y a la aplicación que simula el control de un motor diésel. El objetivo es ver si tras la inyección de un fallo de diseño del software en el sistema, éste se traduce en un error detectado o no y finalmente en una avería en lo que sería la entrega final del servicio.

En primer lugar se va a describir cómo se ha codificado el resultado final de cada experimento. Esta codificación que se va a explicar no hace referencia a la cobertura final del sistema, sino que detalla con respecto a la aplicación, cómo finaliza el experimento. Para obtener datos sobre coberturas habrá que tener en cuenta la detección de errores y otros detalles que más adelante se van a calcular.

De las gráficas siguientes y teniendo en cuenta las posibles salidas al experimento, como se describió en el capítulo anterior, los resultados obtenidos son los siguientes:

Estadísticos

N	Válidos	2000
	Perdidos	0

Tabla 7.1: Resultados de la ejecución

		Frecuencia	Porcentaje	Porcentaje válido	Porcentaje acumulado
Válidos	R0	908	45,4	45,4	45,4
	R1	291	14,6	14,6	60,0
	R2	260	13,0	13,0	73,0
	R3	541	27,1	27,1	100,0
	Total	2000	100,0	100,0	

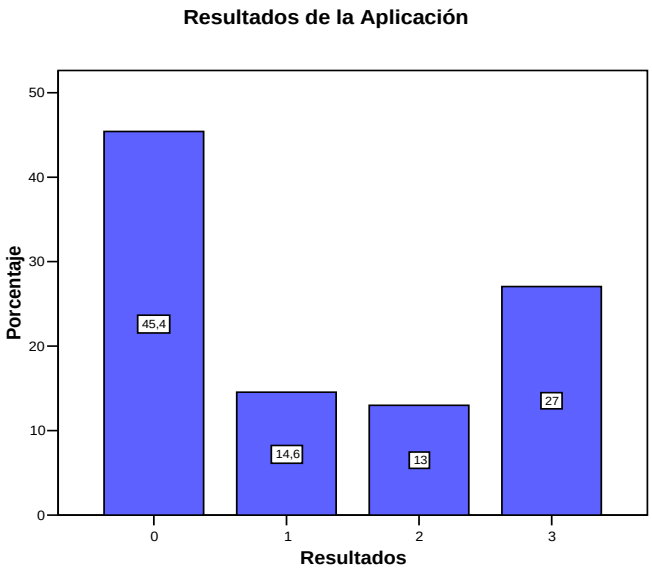


Figura 7.1: Resultados de la ejecución

La consecución del experimento se ha codificado de la siguiente manera:

- **R0**: No hubo error en la entrega del servicio.
- **R1**: El servicio entregado no fue el esperado.
- **R2**: El sistema quedó en una situación de ejecución sin respuesta al exterior (cuelgue del sistema). La aplicación no produce ningún valor de salida.
- **R3**: El microcontrolador produjo una excepción.

Como se puede apreciar en la gráfica de la figura 7.1 y también tabla 7.1 de resultados, se observa que en un 45,4% de las veces se ha obtenido un resultado de "R0", esto se corresponde con aquellos experimentos en los que aún a pesar de haberse producido la inyección efectiva de un fallo, la ejecución de la aplicación y por tanto la entrega final del servicio ofrecido por el sistema no se vieron afectados por la inyección del mismo. En estos casos se puede intuir que los mecanismos intrínsecos del sistema podrían haber actuado o que la inyección podría haberse realizado en localizaciones de memoria que finalmente no son utilizadas por el sistema operativo o arquitectura del microcontrolador, como podrían ser partes altas de palabra de los parámetros de las llamadas o bits no significativos de los parámetros, o también en ocasiones en parámetros que albergan datos devueltos por las llamadas y que por tanto son reescritos. Por tanto en estos casos se puede concluir que la inyección de fallos no afectó al sistema y la ejecución de la aplicación finalizó de un modo correcto.

Cuando se obtuvo un resultado de "R1", que fue en un 14,6% de las veces, en estos casos la inyección del fallo tan sólo afectó a la entrega del servicio, ya que la aplicación se ejecutó normalmente sin excepciones ni cuelgues aunque los resultados obtenidos no fueron correctos, o al menos los esperados, lo cual finalmente supone una avería del sistema. En estos casos haría falta disponer de algún oráculo [Dbench04] que indicara que los valores de salida que se obtienen por parte de la aplicación no son correctos o no están dentro de los rangos permitidos, y así poder detectar este tipo de averías. En el caso que nos ocupa, esto se implementa a través de una doble ejecución con y sin fallos para determinar cuáles son los valores correctos de la ejecución. Esta situación, que se ha denominado como "R1" es bastante crítica, ya que en estos casos parece ser que el sistema está funcionando correctamente pero los resultados que ofrece son incorrectos.

En estos casos para obtener la cobertura final del sistema se tiene que ver si durante esa supuesta correcta ejecución se ha producido alguna notificación por parte del sistema operativo a modo de código de error, que notifique a la aplicación que algo erróneo ha sucedido. Como se verá posteriormente, si hubiera habido casos en los que el SO hubiera producido algún código de error entonces se hubiera podido concluir que el error había sido detectado. En caso de que no se produzca ninguna notificación, se puede hablar de que el sistema se halla ante un fallo crítico del mismo, ya que ninguno de los mecanismos del sistema operativo, ni del microcontrolador, puesto que estos casos se corresponden con los que no se producen tampoco excepciones, han detectado la avería del sistema.

En los casos en los que se obtuvo un resultado codificado como "R2", fueron en aquellas situaciones en las cuales el sistema no estaba ofreciendo claramente ningún servicio, son aquellas ocasiones en las cuales el sistema se halla en una situación de cuelgue, y esto se produjo en un 13% de las veces. Esta situación, que bien podría englobarse dentro de la anterior, puesto que el servicio final que se está ofreciendo es defectuoso o erróneo, ya que como se ha dicho no hay servicio, se ha querido distinguirla del resto porque a diferencia de la anterior, en donde el sistema se encontraba en una situación donde parecía ser que éste funcionaba bien; aquí claramente el sistema no funciona y además no se sabe lo que está haciendo, lo cual supone que se está también ante una avería del sistema. Como en el caso anterior, para obtener la cobertura final del sistema completo y del SO, se tendrá que observar si en esa situación se ha producido algún código de error por parte del SO que permitiera al diseñador de la aplicación recuperar al sistema.

Por último los casos en los cuales se ha obtenido un resultado de "R3", entorno a un 27,1% de las veces, han sido en aquellas ocasiones en las cuales el microcontrolador, con sus mecanismos internos de detección, ha sido el que ha detectado el error y lo ha notificado con el lanzamiento de una excepción y consiguiente parada de la ejecución del sistema. En este caso es el hardware el que ha sido el primero en detectar dicho error.

A continuación en la siguiente tabla 7.2, y con el fin de obtener datos para hallar la cobertura final del sistema y la del propio SO, se va a ver cuantos y cuales han sido los códigos de error (errores detectados) producidos por el SO en cada uno de los experimentos.

Tabla 7.2: Tabla de contingencia Resultado * Código Error del SO

		Códigos de Error							
		0 (No error)	Error 1	Error 40	Error 42	Error 81	Error 82	Error 83	Total
Resultado	R0	908	0	0	0	0	0	0	908
	R1	291	0	0	0	0	0	0	291
	R2	238	8	0	8	1	3	2	260
	R3	532	5	4	0	0	0	0	541
Total		1969	13	4	8	1	3	2	2000

Como se puede observar en la tabla 7.2, se tiene que se han obtenido 908 (45,4%) casos en los cuales ni se ha producido una avería, la ejecución es correcta, ni se ha obtenido ningún código de error. De entre los 291 experimentos en los cuales se ha obtenido un resultado codificado como "R1", es decir una avería del sistema, tampoco se ha producido ningún código de error por parte del SO y por tanto se puede hablar de casos de averías no seguras. En el supuesto de los resultados codificados como "R2", se ha obtenido que de entre un total de los 260 experimentos en los que se obtuvo un cuelgue del sistema, en 22 (8,5%) casos se detectó un error por parte del SO y por tanto estamos ante casos de averías seguras. Y por último, en los casos en los que se produjo una excepción en el sistema en 9 (1,6%) casos de ellos antes del lanzamiento de la excepción se había notificado un error por parte del SO.

Así, se puede concluir que de las averías totales del sistema (R1+R2), que son un total de 551 (27,5%) casos, en 22 (3,8%) casos hubo detección de errores, por tanto averías seguras y en 529 (96%) casos no la hubo, por tanto averías no seguras. Hay que precisar que en el caso de las averías denominadas como seguras, la detección de errores ofrecería, en caso de que así hubiera sido tratado, la posibilidad de que el sistema sea llevado a un estado seguro o a una recuperación del servicio. Así según se puede observar en las gráficas siguientes el porcentaje de averías seguras y no seguras ha sido como sigue:

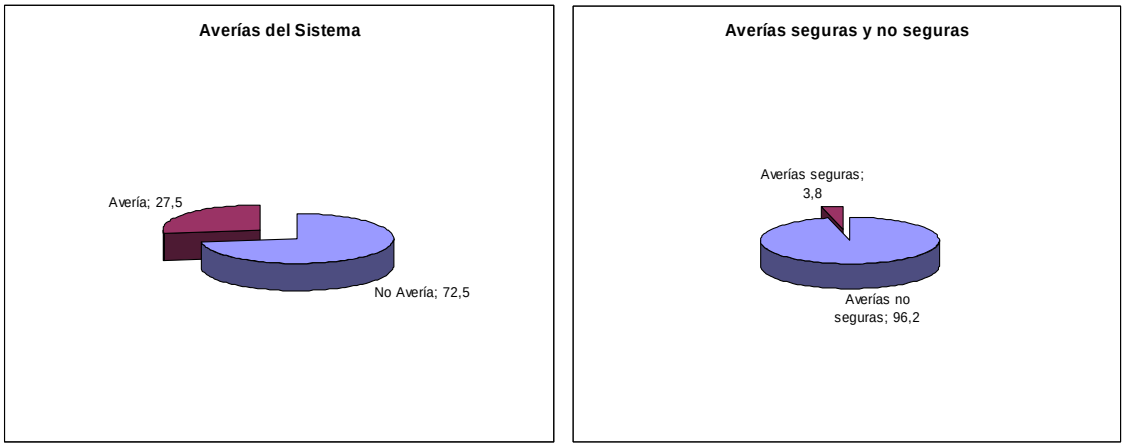


Figura 7.2: Averías del sistema seguras y no seguras

Por tanto, si se define la cobertura total del sistema como aquellas veces en las que éste, constituido por SO, microcontrolador y aplicación ha detectado que un error se ha producido, en forma de código de error o excepción, o por otro lado el sistema ha funcionado correctamente aún a pesar de los fallos inyectados, se tiene que la cobertura final del sistema ha sido:

$$C_{sist.compl.} = R0 + R1(\text{con detección}) + R2(\text{con detección}) + R3 = 908 + 0 + 22 + 541 = 1471\ (73{,}55\ \%)\ \text{casos}$$

Por tanto se puede decir que el 73,5% de las veces el sistema ha cubierto los errores que se ha producido en el mismo, bien ofreciendo un resultado correcto o bien detectando que un error se había producido en el sistema. En estos casos se puede decir que el funcionamiento del sistema podría ser seguro en este porcentaje, si la aplicación tratase correctamente los errores notificados antes de que éstos llegaran a convertirse en averías.

En cuanto a cobertura del sistema operativo se tiene que ésta ha sido como sigue:

$$C_{SO} = R0 + R1(\text{con detección}) + R2(\text{con detección}) = 908 + 0 + 22 = 930\ (46{,}5\ \%)\ \text{casos}$$

En este caso la cobertura del SO vendrá dada por aquellos casos en los que el sistema finalizó la ejecución de la aplicación de un modo correcto, sumado a aquellos casos en los que el propio MicroC/OS-II detectó un error a través de sus propios mecanismos de detección de errores. Por tanto se tiene que el resultado ha sido del 46,45% para la aplicación que se ha evaluado.

7.2.2 Códigos de error devueltos por el SO

En esta sección se describe cuáles han sido los códigos de error devueltos por el SO y su significado, porqué se han producido y qué efecto han tenido en el sistema.

En este caso, para el tratamiento de errores, MicroC/OS-II verifica en las propias llamadas al sistema operativo si se ha producido alguna situación anómala que deba ser tratada con especial atención, o si el servicio que se pide no puede ser ofrecido. En tal caso, la llamada al sistema operativo devolverá un código de error en el valor de retorno de la función o en uno de los argumentos de la llamada como un entero de 8 bits sin signo. Los códigos de error soportados por MicroC/OS-II se pueden encontrar en el fichero "ucos_ii.h" y contiene un total de 31 tipos diferentes de códigos de error.

En la siguiente tabla 7.3, se pueden ver cuales han sido y en que porcentaje se han obtenido códigos de error tras la ejecución de los experimentos de inyección de fallos:

Tabla 7.3: Códigos de Error

		Frecuencia	Porcentaje	Porcentaje válido	Porcentaje acumulado
Válido	Error 1	13	41,9	41,9	41,9
	Error 40	4	12,9	12,9	54,8
	Error 42	8	25,8	25,8	80,6
	Error 81	1	3,2	3,2	83,9
	Error 82	3	9,7	9,7	93,5
	Error 83	2	6,5	6,5	100,0
	Total	31	100,0	100,0	

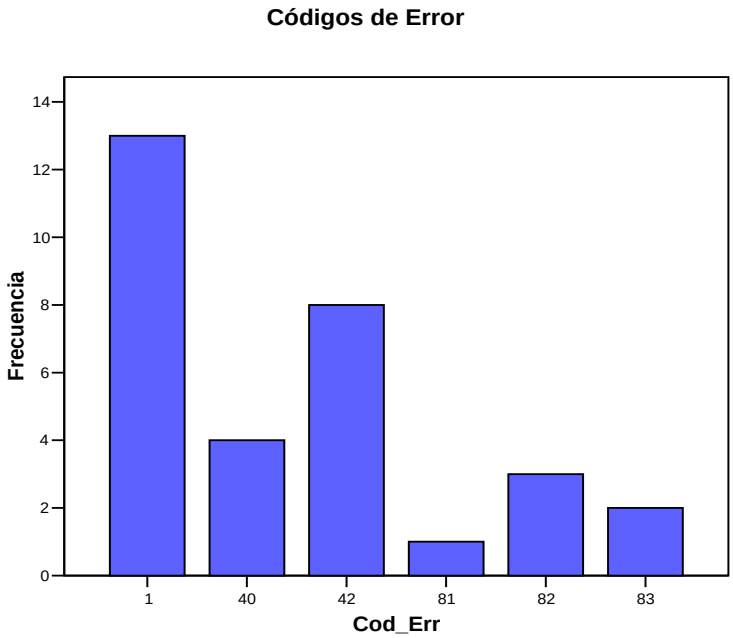


Figura 7.3: Frecuencia de obtención de los Códigos de Error

Clasificación y descripción de los códigos de error obtenidos:

- **Error 1**: Este código de error se codifica como "OS_ERR_EVENT_TYPE". Está relacionado con estructuras de tipo "OS_EVENT", que sirven para apuntar a colas de mensajes, semáforos, buzones y otros mecanismos de sincronización del SO. Se ha obtenido en un 41,9% de las veces. Llamadas relacionadas con este código de error son: OSMboxPend, OSMboxPost, OSMboxQuery, OSQFlush, OSQPend, OSQPost, OSQPostFront, OSQQuery, OSSemPend, OSSemPost y OSSemQuery. Este código de error se produce cuando el argumento "OS_EVENT *pevent" de estas llamadas al sistema no apunta correctamente a uno de los mecanismos de sincronización, por supuesto dependiendo de la llamada que se ha mencionado anteriormente.

- **Error 40**: Este código de error se codifica como “OS_PRIO_EXIST”. Está relacionado con la creación de nuevas tareas en el sistema o con la asignación dinámica de prioridades a tareas ya existentes en el mismo. Se ha obtenido en un 12,9% de las veces y se produce cuando se intenta asignar a una tarea nueva o existente, un valor de prioridad que ya existe para otra tarea que está funcionando en el sistema. Llamadas relacionadas con este código de error son: OSTaskChangePrio, OSTaskCreate y OSTaskCreateExt.

- **Error 42**: Este código de error se codifica como “OS_PRIO_INVALID”. También está relacionado con la creación de nuevas tareas en el sistema o con la asignación dinámica de prioridades a tareas ya existentes en el mismo. Este error se produce cuando se intenta asignar un valor para la prioridad que se sale de rango o el valor nuevo a asignar es idéntico al que la tarea ya posee. Se ha obtenido en un 25,8% de las veces. Llamadas relacionadas con este código de error son: OSTaskChangePrio, OSTaskCreate, OSTaskCreateExt, OSTaskDel, OSTaskDelReq, OSTaskQuery, OSTaskResume, OSTaskSuspend y OSTimeDlyResume.

- **Error 81**: Este código de error se codifica como “OS_TIME_INVALID_MINUTES”. Está relacionado con la llamada al sistema OSTimeDlyHMSM, que sirve para que una tarea se retarde a si misma un periodo de tiempo especificado en horas, o minutos, o segundos o incluso milisegundos. Se ha obtenido en un 3,2% de las veces y significa que el contenido del parámetro que hace referencia a los minutos que la tarea debe retardarse es incorrecto.

- **Error 82**: Este código de error se codifica como “OS_TIME_INVALID_SECONDS”. Está relacionado con la misma llamada al sistema que en el anterior caso, pero esta vez diciendo que los segundos especificados en el parámetro correspondiente de la llamada al sistema OSTimeDlyHMSM son incorrectos. Se ha obtenido en un 9,7% de las veces.

- **Error 83**: Este código de error se codifica como “OS_TIME_INVALID_MILLI”. También relacionado con la misma llamada al sistema que en el anterior caso, pero esta vez diciendo que los milisegundos especificados en el parámetro correspondiente de la llamada al sistema OSTimeDlyHMSM son incorrectos. Se ha obtenido en un 6,5% de las veces.

7.2.3 Excepciones

Una vez se han estudiado los códigos de error obtenidos por parte del SO, a continuación se va a analizar cuáles han sido las excepciones lanzadas por el microcontrolador MPC565, que se corresponde con aquellos casos en los que tras la inyección de un fallo, los mecanismos de detección de errores del microcontrolador son los que detectan un error en el sistema y en este caso lo llevan a finalizar la ejecución. Estos han sido codificados como salida “R3” del experimento y han supuesto un 27,1% del total de los experimentos.

En la figura 7.4 se puede observar qué excepciones se han obtenido y en qué porcentaje. Como se puede ver el mecanismo que más errores ha detectado es la detección de código de operación incorrecto, activando la excepción de emulación software (SEE) al intentar ejecutar instrucciones de acceso a la caché de datos o a registros de segmento, o instrucciones de acceso a registros de propósito especial.

Otro dato interesante también es que un porcentaje alto de las detecciones de error se corresponden con excepciones de tipo “*machine check*” (MCE). Estas excepciones no

garantizan que se conserve el estado de los registros del procesador y, por lo tanto, se consideran no recuperables. En estos, casos la única forma de recuperar el sistema sería mediante un reset. En segundo lugar y con un porcentaje muy similar la excepción de "*checkstop*" (CHSTP) que se produce tras una excepción de "*machine check*", con lo que una es consecuencia de la otra, donde también se produce una suspensión en el procesamiento de instrucciones y el sistema debe ser reseteado para ser recuperado. En estos casos esto puede ser debido por al acceso a una dirección que no existe, o un error en los datos, o una violación de protección de almacenamiento.

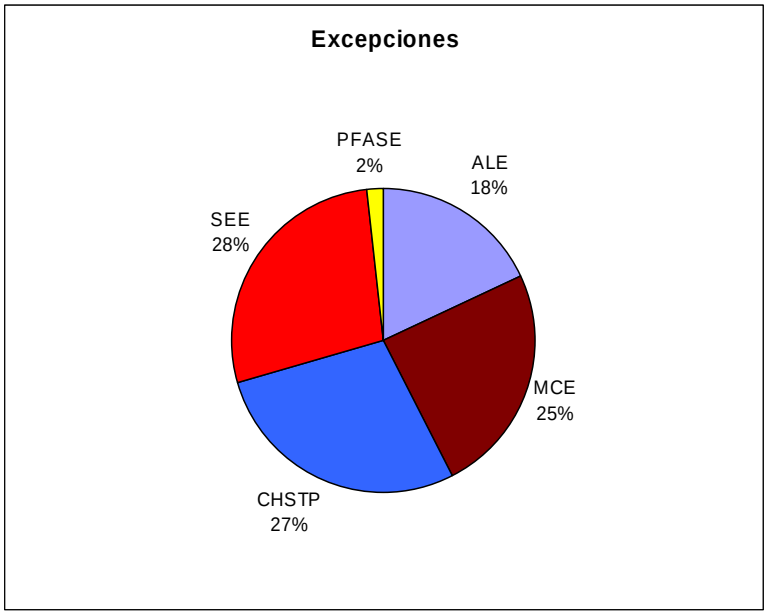


Figura 7.4: Frecuencia de obtención de las Excepciones

También se tiene que en un porcentaje correspondiente al 18% se han obtenido errores indicados con la excepción ALE, que son errores de alineamiento en la memoria, probablemente provocados por una mutación en algún operando accedido en modo indirecto. Y en último lugar, las excepciones FPASE que indican errores detectados en la unidad de coma flotante y que se han producido en muy pocos casos.

7.2.4 Tiempos

En el siguiente apartado se va estudiar cómo las campañas de inyección de fallos han podido incidir en los resultados temporales que se esperan de un funcionamiento correcto del sistema. Como se ha adelantado en los primeros capítulos, las averías del sistema se pueden producir tanto por la obtención de valores incorrectos durante la ejecución del sistema, y por tanto se habla de averías en el dominio del valor, como por averías en el dominio del tiempo, por la obtención de resultados fuera de los límites temporales marcados como "*deadlines*" para la obtención de dichos valores correctos. Por tanto, a continuación se va a examinar los tiempos de detección de errores del SO, los tiempos de las tareas de la aplicación en ejecución libre de errores, y la influencia de los fallos inyectados en los tiempos de ejecución de las mismas.

7.2.4.1 Tiempos de detección del SO

En este apartado se van a calcular los tiempos de detección de errores del SO, siguiendo la metodología anteriormente propuesta para la obtención de tiempos de latencia de detección.

Para ello se obtiene la diferencia temporal entre el instante en el que el fallo se hace efectivo (se activa) en el sistema y el instante en el cual el mecanismo de detección de errores del SO en cuestión notifica un código de error indicando un estado erróneo del sistema. Así por tanto los datos obtenidos son (valores en microsegundos):

Tabla 7.4: Tiempos de detección de errores del SO

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Cod_Err 1	13	12.200	13.800	12.81538	.224707	.810191	,656
N válido (según lista)	13						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Cod_Err 40	4	21.960	22.100	22.06500	.035000	.070000	,005
N válido (según lista)	4						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Cod_Err 42	8	12.800	12.800	12.80000	.000000	.000000	,000
N válido (según lista)	8						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Cod_Err 81	1	23.760	23.760	23.76000	.	.	.
N válido (según lista)	1						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Cod_Err 82	3	23.760	23.760	23.76000	.000000	.000000	,000
N válido (según lista)	3						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Cod_Err 83	1	23.760	23.760	23.76000	.	.	.
N válido (según lista)	1						

En la tabla 7.4 se pueden observar los tiempos medios, valores máximos y mínimos obtenidos con el objetivo de calcular, cuál es el coste temporal que supone tener una notificación por parte del SO de que un error ha sido detectado durante la ejecución de la aplicación. De los estadísticos anteriores destacar que en este caso, y para la aplicación dada, los errores relacionados con la creación de tareas, asignación y cambio de prioridades de las mismas, y los errores relacionados con llamadas al sistema que manejan aspectos de temporización del sistema, son los que han tenido un coste temporal mayor, aunque y como se puede ver en las siguientes figura 7.5, los tiempos de detección de errores de éstos están muy próximos en todos los casos.

A continuación y de un modo no tan específico se van a analizar los datos totales sobre los tiempos de detección de errores del SO en todos los casos. Así por tanto se tiene que (valores en microsegundos):

Estadísticos

N	Válidos	30
	Perdidos	0
Media		15.86867
Error típico de la media		.877144
Mediana		12.80000
Moda		12.200(a)
Desviación típica		4.804316
Varianza		23,081
Mínimo		12.200
Máximo		23.760

(a) Existen varias modas. Se mostrará el menor de los valores.

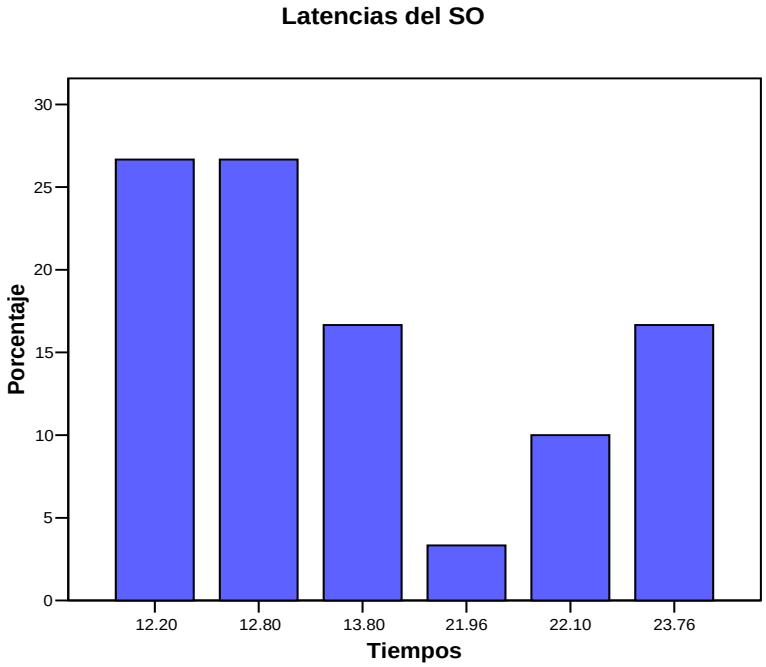


Figura 7.5: Frecuencia de las latencias de detección

Como se puede extraer de la figura 7.5 los tiempos de detección de errores del SO, para la aplicación dada, están entorno a una media de 15,8us, aunque como se puede ver en esta última gráfica en más del 50% de casos las latencias has oscilado entre los 12 y los 13 microsegundos. Por tanto, viendo la distribución de tiempos que muestra la figura 7.5, se puede decir que las latencias de detección se centran más en dicha franja de tiempos, al ser éstos los tiempos de latencia que con mayor frecuencia se han obtenido y es por ello que la mediana está entorno a los 12,8us y la menor de las modas entorno a los 12,2us.

7.2.4.2 Tiempos de las Tareas

A continuación en este apartado se va mostrar cual ha sido el efecto de los fallos inyectados con respecto a los tiempos de ejecución de las tareas. El objetivo es poder observar cómo la inyección de fallos de diseño del software, en un sistema constituido por componentes COTS, puede afectar a los propios componentes en el cumplimiento del trabajo asignado a cada una de las tareas del sistema dentro de los tiempos establecidos. Con esto se puede ver que a pesar de que los valores obtenidos por parte de la aplicación al final son correctos, los tiempos en los que

estos valores son producidos pueden variar. Estas variaciones temporales a las que se hace referencia, podrán ser más o menos críticas dependiendo del criterio del diseñador de la aplicación, a la hora de establecer los tiempos máximos asignados a las tareas, y que delimitarían las averías del sistema en el dominio del tiempo. En la siguiente tabla 7.5, se pueden ver las diferentes tareas que se ejecutan en el sistema, su función y el nombre que se les ha dado a la hora de identificar los tiempos obtenidos.

Tabla 7.5: Descripción de las tareas del sistema.

Nombre	Tarea	Descripción
Tarea1	TareaControlPresionRail	Bucle de control de la presión del raíl
Tarea2	TareaControlPresionAdmision	Bucle de control de la presión de admisión
Tarea3	TareaControlSWIRL	Bucle de control de las válvulas SWIRL
Tarea4	TareaCalculoConsignas	Bucle de control de la temporización de la inyección
Tarea5	TareaRecepcionCAN	Bucle de control de recepción de mensajes CAN
Tarea6	TareaPrincipal	Bucle de control de la monitorización del Estado

Los siguientes estadísticos representan en primer lugar los datos obtenidos correspondientes a los tiempos máximos de ejecución de las tareas para aquellos experimentos en los que no se inyectaban fallos (ejecuciones libres de fallos). En estos casos, dado que los resultados de los experimentos son deterministas se han tomado un conjunto muestras suficiente para establecer la referencia temporal de las tareas en ejecución libre de fallos. Seguidamente se muestran los resultados obtenidos para los tiempos máximos de ejecución de las tareas de la aplicación, pero una vez se han inyectado fallos en el sistema. En estos casos los tiempos que se obtienen se corresponden con aquellos experimentos que se han codificado como "R0", donde se observaba que la inyección de un fallo finalmente no afectaba, desde el punto de vista del dominio del valor, a la entrega del servicio. Así por tanto, lo tiempos obtenidos tarea por tarea son los siguientes (valores en milisegundos):

Tabla 7.6: Tiempos para la Tarea1 sin fallos y con fallos

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Tarea1	98	3,137	3,147	3,14333	,000489	,004846	,000
N válido (según lista)	98						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Tarea1	590	3,128	5,758	3,18823	,013847	,336353	,113
N válido (según lista)	590						

Como se ha dicho, en primer lugar se presentan los datos correspondientes a las ejecuciones libres de errores y en segundo lugar los datos correspondientes a los experimentos donde se han inyectado fallos. En este caso para la Tarea1 ("ControlPresionRail") se observa que la media para los tiempos máximos en presencia de fallos se incrementa en unos 45us, aunque lo más significativo es que la tarea puede llegar a obtener unos tiempos de respuesta de 5,76ms, es decir 2,6ms de más al tiempo máximo obtenido en la ejecución libre de errores, casi duplicando su tiempo máximo de ejecución y posiblemente generando averías por entrega tardía del servicio, aunque como se ha comentado anteriormente esto dependerá de las especificaciones que el diseñador del sistema tenga.

Tabla 7.7: Tiempos para la Tarea2 sin fallos y con fallos

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Tarea2	98	2,848	2,848	2,84800	,000000	,000000	,000
N válido (según lista)	98						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Tarea2	590	,087	19,813	2,92970	,046574	1,131280	1,280
N válido (según lista)	590						

Para el caso de la Tarea2 (“TareaControlPresionAdmision”) se observa que la media para los tiempos máximos en presencia de fallos se incrementa entorno a los 90us. Y lo que más llama la atención en este caso es que se han producido valores mínimos de ejecución entorno a los 87us, cuando la tarea debe ejecutarse entorno a los 2,8ms, pudiendo producir en este caso averías por entrega temprana del servicio. En cuanto a los valores máximos se puede ver que ha habido casos en los cuales casi se ha llegado a los 20ms, multiplicando por 10 el tiempo de ejecución habitual de la tarea.

Tabla 7.8: Tiempos para la Tarea3 sin fallos y con fallos

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Tarea3	98	,351	2,987	1,26724	,125745	1,244810	1,550
N válido (según lista)	98						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Tarea3	590	,351	2,987	1,76290	,053474	1,298873	1,687
N válido (según lista)	590						

Para el caso de la Tarea3 (“TareaControlSWIRL”) se observa que la media para los tiempos máximos en presencia de fallos se incrementa entorno a los 50us. Aunque como se puede ver los tiempos máximos y mínimos están dentro del rango de valores normales en los que dicha tarea se ha ejecutado en los experimentos libres de errores. Por tanto se puede asegurar que en este caso la inyección de los fallos no ha tenido prácticamente efecto sobre esta tarea con respecto a sus tiempos de ejecución.

Tabla 7.9: Tiempos para la Tarea4 sin fallos y con fallos

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Tarea4	98	4,438	4,474	4,46314	,001470	,014556	,000
N válido (según lista)	98						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Tarea4	590	4,438	7,090	4,49656	,011586	,281413	,079
N válido (según lista)	590						

Para el caso de la Tarea4 (“TareaCalculoConsignas”) se observa que la media para los tiempos máximos en presencia de fallos se incrementa entorno a los 40us. Los tiempos mínimos están dentro de los valores normales pero los tiempos máximos obtenidos puede llegar a alcanzar los 7ms, elevando considerablemente en algunos casos el tiempo que la tarea ha requerido para llevar a cabo sus labores.

Tabla 7.10: Tiempos para la Tarea5 sin fallos y con fallos

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Tarea5	98	2,803	2,803	2,80300	,000000	,000000	,000
N válido (según lista)	98						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Tarea5	590	2,803	5,424	2,81635	,007681	,186578	,035
N válido (según lista)	590						

Para el caso de la Tarea5 (“TareaRecepcionCAN”) se observa que la media para los tiempos máximos en presencia de fallos no se incrementa significativamente. Aunque para los tiempos máximos obtenidos se puede llegar a alcanzar los 5,4ms, elevando en este caso en 2,6ms el tiempo que en algunos experimentos la tarea ha necesitado para realizar su trabajo.

Tabla 7.11: Tiempos para la Tarea6 sin fallos y con fallos

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Tarea6	98	17,667	17,685	17,67422	,000554	,005488	,000
N válido (según lista)	98						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Tarea6	590	1,05	20,31	17,7710	,05982	1,45295	2,111
N válido (según lista)	590						

Y en última instancia se tiene el caso de la Tarea6 (“TareaPrincipal”) que se encarga del bucle de control de monitorización de la variable de estado, y en definitiva de producir los valores de salida de la ECU. En este caso se observa que la media con fallos difiere en unos 100us. Pero lo más significativo es que este caso la tarea puede producir resultados en 1ms y en 20ms. Si consideramos que el rango normal de operación para dicha tarea está entre los [17,667ms, 17,685ms], se ve claramente que en algunos experimentos se han producido tanto averías por entrega temprana como averías por entrega tardía de los valores de salida de la unidad de control electrónico del motor diésel.

7.2.4.3 Distribución de los tiempos máximos

En al apartado anterior se han estudiado los valores obtenidos de los estadísticos producidos a partir de la evaluación de los tiempos máximos de ejecución de las tareas. A continuación se va analizar, cuál es la distribución de esos tiempos máximos para cada una de las tareas en los experimentos que se han llevado a cabo. El objetivo es poder observar cual de éstas se ha visto mayormente afectada en sus tiempos de ejecución a causa de los fallos introducidos y cuales de

éstas serían más propensas a introducir retardos o adelantos en el tiempo total de ejecución del sistema.

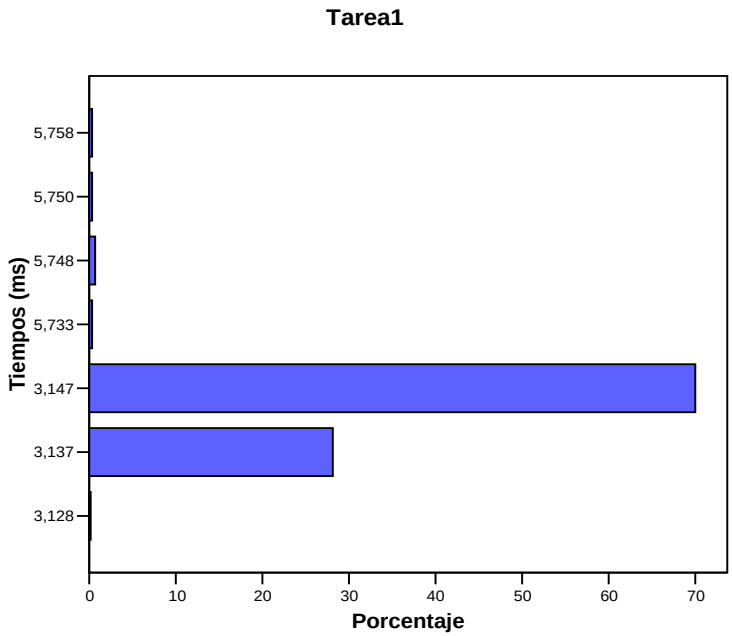


Figura 7.6: Distribución de los tiempos máximos para la Tarea1

En primer lugar se estudiarán los datos relativos a la distribución de los tiempos máximos de ejecución obtenidos para la Tarea1 (“ControlPresionRail”). Tal como se puede observar en la figura 7.6 se tiene que cerca del 70% de los experimentos se ha obtenido un tiempo de ejecución para la tarea de 3,147ms y próximo al 27% un tiempo de 3,137ms. Si se tiene que el rango de valores normales de ejecución de dicha tarea está entre [3,137ms, 3,147ms] en principio se puede concluir que la Tarea1 no se ha visto muy afectada por fallos inyectados. Aunque sí que se observa que dentro de aquellos casos en los que no se han cumplido los tiempos de ejecución esperados para la tarea, se han producido más casos de entrega tardía que temprana del servicio, incrementándose dichos tiempos de ejecución para la tarea entorno a los 2,5ms.

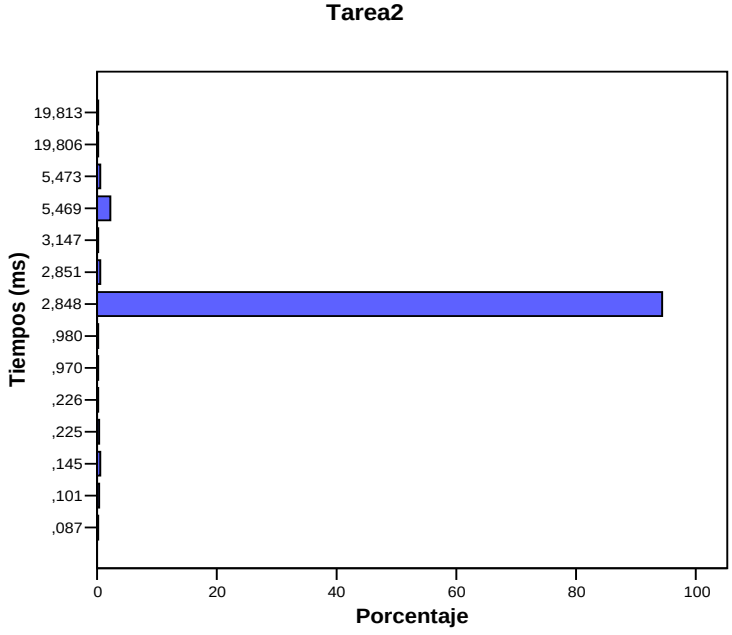


Figura 7.7: Distribución de los tiempos máximos para la Tarea2

Para la Tarea2 (“TareaControlPresionAdmision”) en este caso se tiene que en más del 90% de los experimentos se ha obtenido un tiempo de 2,848ms, que es el tiempo que debe cumplir la tarea en condiciones normales. Para esta tarea, si que hay una distribución casi a la par entre averías por entrega tardía del servicio como por pronta entrega. Destacar, como se ha comentado anteriormente, los valores extremos obtenidos en ambos sentidos que pueden afectar cuantitativamente al tiempo de ejecución final resultante del sistema.

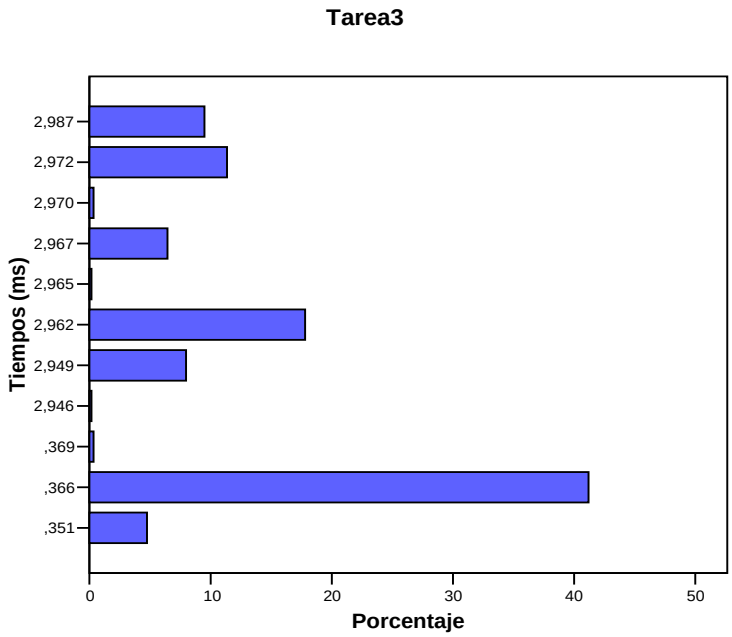


Figura 7.8: Distribución de los tiempos máximos para la Tarea3

Para la Tarea3 (“TareaControlSWIRL”) se tiene que la variabilidad en los tiempos de ejecución de dicha tarea es bastante elevada. Se observa que más del 40% de las veces se ha recogido un tiempo de ejecución de 0,366ms y cerca del 60% valores entorno a los 3ms. Aunque, este caso hay que decir que todos los tiempos que se muestran en la gráfica entran dentro del rango de tiempos normales para dicha tarea.

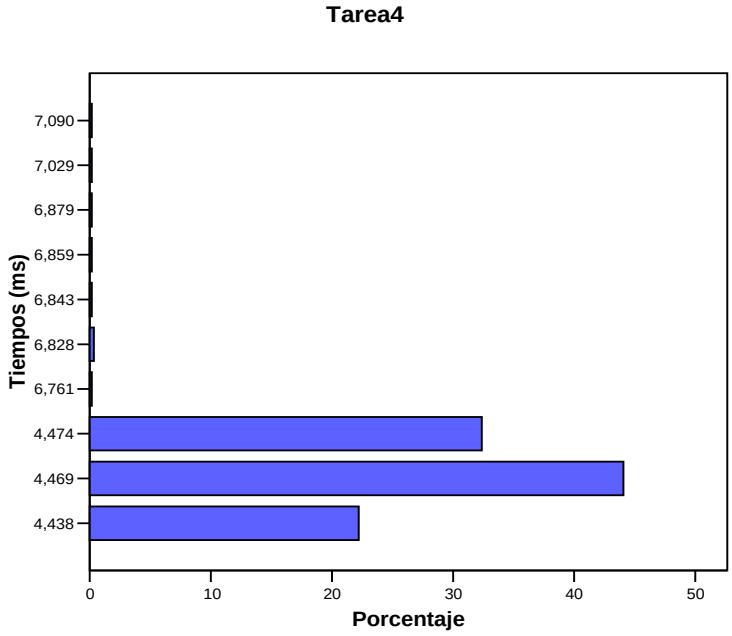


Figura 7.9: Distribución de los tiempos máximos para la Tarea4

Para la Tarea4 (“TareaCalculoConsignas”) se tiene que en cerca del 90% de los experimentos se han obtenido tiempos entorno a los 4,4ms, que es el tiempo que debe cumplir la tarea en condiciones normales. Destacar que en este caso, las averías vienen por una entrega tardía del servicio que en la mayoría de los casos se produce a partir de los 6,7ms que tardaría en ejecutarse dicha tarea.

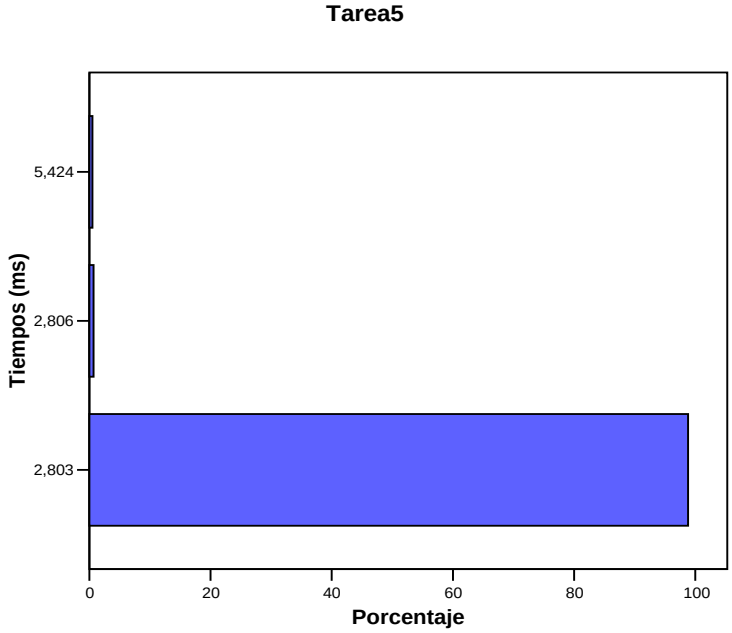


Figura 7.10: Distribución de los tiempos máximos para la Tarea5

Para la Tarea5 (“TareaRecepcionCAN”) se observa que la distribución de los tiempos de ejecución ha sido bastante uniforme entorno a los 2,8ms, por tanto se puede afirmar que dicha tarea prácticamente no se ha visto afectada por los fallos inyectados, aproximándose en la mayoría de los experimentos a sus tiempos normales de ejecución libre de errores. Sí que hay algunos experimentos en los cuales se ha alcanzado los 5,4ms aunque más bien son pocos casos.

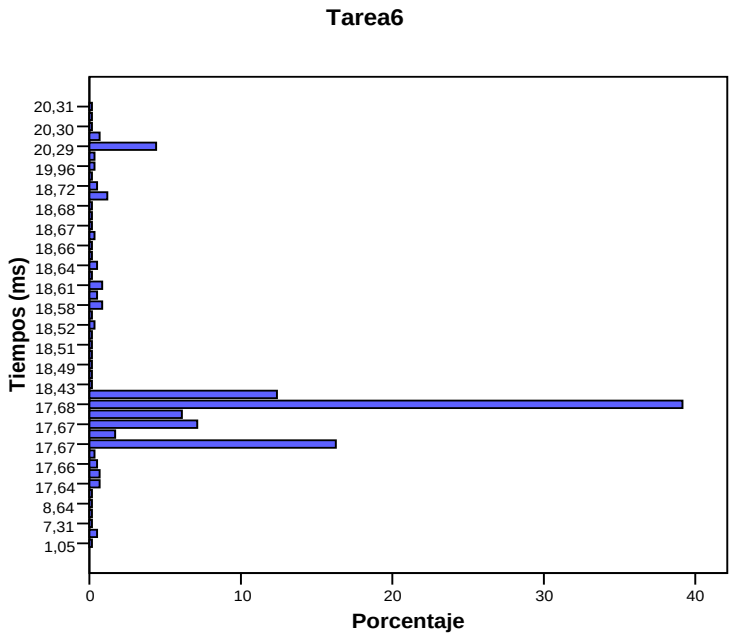


Figura 7.11: Distribución de los tiempos máximos para la Tarea6

Y en última instancia, con respecto a los datos relativos a la Tarea6 ("TareaPrincipal"), que es la tarea encargada de producir los valores de salida de estado de la ECU. En este caso alrededor de un 75% de los tiempos de ejecución obtenidos se hallan dentro del rango de tiempos entorno a los [17,667ms, 17,685ms], que se corresponde con los valores normales de operación de la tarea en ejecución libre de fallos. Ello supone que aproximadamente un 25% de los casos, los valores de los datos de salida de la aplicación se han producido en tiempos por encima o por debajo de los valores normales. Como se puede observar en la gráfica de la figura 7.11, se producen más casos de entregas tardías del servicio que averías por adelanto. En resumen se puede afirmar que como la "TareaPrincipal" se nutre de los resultados provenientes del resto de tareas, es la que mayor variabilidad presenta en sus tiempos de ejecución, así como la más susceptible de verse perjudicada y producir más averías en el sistema por el retardo o adelanto de la entrega de datos que espera por parte del resto de tareas.

7.3 RESULTADOS PARA OSEK/VDX Y EL CONTROL DE SEMÁFOROS

A continuación y siguiendo la misma estructura que en el apartado anterior, así el lector podrá recurrir a los datos de cada una de las aplicaciones testeadas de forma independiente, se van a presentar los resultados obtenidos para una carga constituida por una implementación de la especificación del sistema operativo de tiempo real OSEK/VDX y una aplicación de un control de semáforos sensorizados para un cruce de dos calles con tráfico en ambos sentidos[6].

Posteriormente, se presentan los resultados obtenidos para la misma aplicación de control de semáforos, pero esta vez funcionando sobre otro RTOS como es MicroC/OS-II. El objetivo es ver para una aplicación dada, que RTOS se ajustaría mejor a las especificaciones que se tuvieran con respecto a la confiabilidad final del sistema y así tener datos objetivos sobre la idoneidad de utilizar uno u otro SO.

7.3.1 Coberturas

En este apartado se va a estudiar cuáles han sido los resultados obtenidos en cuanto a coberturas de detección de errores tanto del propio sistema operativo bajo estudio, en este caso OSEK/VDX, como la cobertura final del sistema, que integrará además los datos obtenidos del propio microcontrolador MPC565 y la aplicación que simula el control de semáforos. Se han realizado un total de 1792 experimentos y estos son los resultados obtenidos:

Estadísticos

N	Válidos	1792
	Perdidos	0

Tabla 7.12: Resultados de la ejecución

		Frecuencia	Porcentaje	Porcentaje válido	Porcentaje acumulado
Válidos	R0	733	40,9	40,9	40,9
	R1	477	26,6	26,6	67,5
	R2	525	29,3	29,3	96,8
	R3	57	3,2	3,2	100,0
	Total	1792	100,0	100,0	

[6] Dicha carga ha sido descrita en el capítulo anterior.

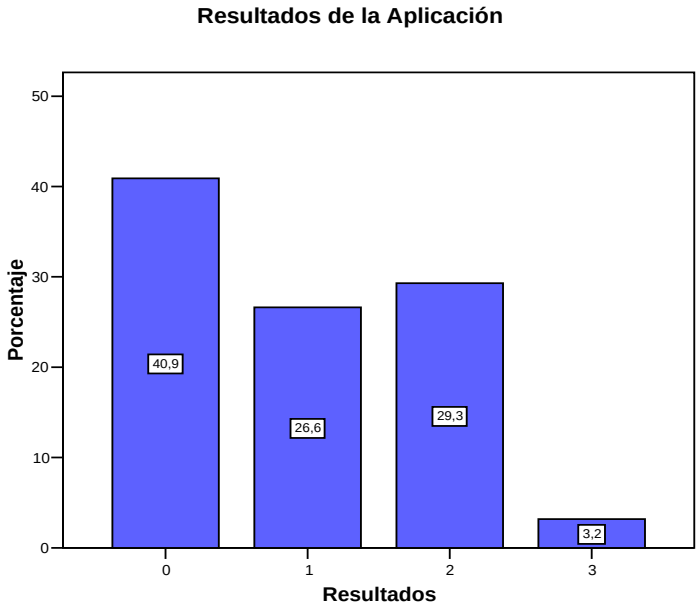


Figura 7.12: Resultados de la ejecución

En primer lugar se va a describir, al igual que anteriormente se hizo, como se ha codificado el resultado final de cada experimento. Esta codificación que se va a explicar no hace referencia a la cobertura final del sistema, sino que detalla con respecto a la aplicación, cómo finaliza el experimento. Para obtener datos sobre coberturas habrá que tener en cuenta la detección de errores y otros detalles que más adelante se van a calcular.

La consecución del experimento se ha codificado de la siguiente manera:

- **R0**: No hubo error en la entrega del servicio.
- **R1**: El servicio entregado no fue el esperado.
- **R2**: El sistema quedó en una situación de ejecución sin respuesta al exterior (cuelgue del sistema). La aplicación no produce ningún valor de salida.
- **R3**: El microcontrolador produjo una excepción.

Acorde a esto y tal y como se puede apreciar en la gráfica y tabla de resultados de la figura 7.12, se observa que en un 40,9% de las veces se ha obtenido un resultado de “R0”, esto se corresponde con aquellos experimentos en los que aún a pesar de haberse producido la inyección efectiva de un fallo, la ejecución de la aplicación y por tanto la entrega final del servicio ofrecido por el sistema no se vieron afectados por la inyección de dicho fallo. En estos casos se puede intuir que los mecanismos de tolerancia a fallos del sistema podrían haber actuado o que la inyección podría haberse realizado en localizaciones de memoria que finalmente no son utilizadas por el sistema operativo o arquitectura del microcontrolador, como podrían ser partes altas de palabra de los parámetros de las llamadas, o bits no significativos de los parámetros, o también en ocasiones en parámetros que albergan datos devueltos por las llamadas y que por tanto son reescritos. Por tanto en estos casos se puede concluir que la inyección de fallos no afectó al sistema y la ejecución de la aplicación finalizó de un modo correcto.

Cuando se obtuvo un resultado de “R1”, que fue en un 26,6% de las veces, en estos casos la inyección del fallo tan sólo afectó a la entrega del servicio, ya que la aplicación se ejecutó normalmente sin excepciones ni cuelgues aunque los resultados obtenidos no fueron correctos, o al menos los esperados, lo cual finalmente supone una avería del sistema. En estos casos haría falta disponer de algún oráculo [Dbench04] que indicara que los valores de salida que se

obtienen por parte de la aplicación no son correctos o no están dentro de los rangos permitidos, y así poder detectar este tipo de averías. En el caso que nos ocupa, esto se implementa a través de una doble ejecución con y sin fallos para determinar cuáles son los valores correctos de la ejecución. Esta situación, que se ha denominado como "R1" es bastante crítica, ya que en estos casos parece ser que el sistema está funcionando correctamente pero los resultados que ofrece son incorrectos.

En estos casos para obtener la cobertura final del sistema se tiene que ver si durante esa supuesta correcta ejecución se ha producido alguna notificación por parte del sistema operativo a modo de código de error, que notifique a la aplicación que algo erróneo ha sucedido. Como se verá posteriormente, si hubiera habido casos en los que el SO hubiera producido algún código de error entonces se hubiera podido concluir que el error había sido detectado. En caso de que no se produzca ninguna notificación, se puede hablar de que el sistema se halla ante un fallo crítico del mismo, ya que ninguno de los mecanismos del sistema operativo, ni del microcontrolador, puesto que estos casos se corresponden con los que no se producen tampoco excepciones, han detectado la avería del sistema.

En los casos en los que se obtuvo un resultado codificado como "R2", fueron en aquellas situaciones en las cuales el sistema no estaba ofreciendo claramente ningún servicio, son aquellas ocasiones en las cuales el sistema se halla en una situación de cuelgue, y esto se produjo en un 29,3% de las veces. Esta situación, que bien podría englobarse dentro de la anterior, puesto que el servicio final que se está ofreciendo es defectuoso o erróneo, ya que como se ha dicho no hay servicio, se ha querido distinguirla del resto porque a diferencia de la anterior, en donde el sistema se encontraba en una situación donde parecía ser que éste funcionaba bien; aquí claramente el sistema no funciona y además no se sabe lo que está haciendo, lo cual supone que se está también ante una avería del sistema. Como en el caso anterior, para obtener la cobertura final del sistema completo y del SO, se tendrá que observar si en esa situación se ha producido algún código de error por parte del SO que permitiera al diseñador de la aplicación recuperar al sistema.

Por último los casos en los cuales se ha obtenido un resultado de "R3", entorno a un 3,2% de las veces, han sido en aquellas ocasiones en las cuales el microcontrolador, con sus mecanismos internos de detección, ha sido el que ha detectado el error y lo ha notificado con el lanzamiento de una excepción y consiguiente parada de la ejecución del sistema. En este caso es el hardware el que ha sido el primero en detectar dicho error.

A continuación en la siguiente tabla 7.13, y con el fin de obtener datos para hallar la cobertura final del sistema y la del propio SO, se va a ver cuantos y cuales han sido los códigos de error (errores detectados) producidos por el SO en cada uno de los experimentos.

Tabla 7.13: Tabla de contingencia de Resultado * Código Error del SO.

		Códigos de Error							
		Error 4101	Error 4301	Error 5301	Error 5302	Error 5501	Error 5502	Error 6101	Total
Resultado	R0	0	0	0	0	0	733	0	733
	R1	0	0	0	0	0	477	0	477
	R2	198	37	78	26	88	82	16	525
	R3	0	0	0	0	0	57	0	57
Total		198	37	78	26	88	1349	16	1792

De la tabla 7.13 se puede observar que para los casos clasificados como "R0" se han obtenido 733 casos con el código 5502, que no es un error sino un aviso o "*warning*" del SO que se produce al inicio de la aplicación notificando que hay una alarma que aún no está activada. Estos casos se engloban en aquellas situaciones que se describían en el capítulo anterior en las que se puede esperar obtener un código de error por parte del SO, sin que esto

suponga que algo está funcionando incorrectamente, ya que en este caso el código de error es esperado. Por tanto los 733 (40,9%) casos se corresponden con un funcionamiento normal o correcto del sistema.

Para los casos de "R1" aparte de los *warnings*, que se han comentado anteriormente, no se ha obtenido ningún código de error adicional que realmente signifique que el SO ha detectado un error, por tanto se puede decir que los 477 (26,6%) casos se corresponden con averías no seguras del sistema, ya que al final la aplicación no ha funcionado correctamente y no se ha detectado ningún error. Significativos son los resultados para "R2", donde como se puede apreciar en la tabla 7.13 se han obtenido 443 códigos de error; es decir un 84,4% de la veces que el sistema se ha colgado se ha recibido por parte del SO una notificación a modo de código de error, en este caso se habla de averías seguras. Esto hace por tanto, que dichos errores hayan sido cubiertos por el propio SO, dando así la posibilidad al diseñador del sistema de activar algún mecanismo adicional para recuperar el servicio, ya que como también se ha comentado anteriormente, en estos casos el sistema no se sabe que está haciendo. Por tanto han habido 82 cuelgues efectivos del sistema de los experimentos totales, es decir un 15,6% de las veces el sistema se ha colgado sin posibilidad de recuperación con los mecanismos propios del SO, se habla también en este caso de averías no seguras del sistema. Para finalizar los casos codificados como "R3", donde se ha finalizado la ejecución como consecuencia del lanzamiento de una excepción por parte del microcontrolador, no se ha emitido ningún código de error por parte del SO y por tanto es el microcontrolador el que primero que ha detectado y cubierto el error en el sistema.

Así por tanto, se puede concluir que de las averías totales del sistema (R1+R2) éstas constituyen un total de 927 (55,9%) casos, donde en 443 (47,8%) casos hubo detección de errores, por tanto averías seguras y en 484 (52,2%) casos no la hubo, por tanto averías no seguras. Hay que precisar que en el caso de las averías denominadas como seguras, la detección de errores ofrecería, en caso de que así hubiera sido tratado, la posibilidad de que el sistema sea llevado a un estado seguro o a una recuperación del servicio. Así según se puede apreciar en las gráficas siguientes el porcentaje de averías seguras y no seguras ha sido como sigue:

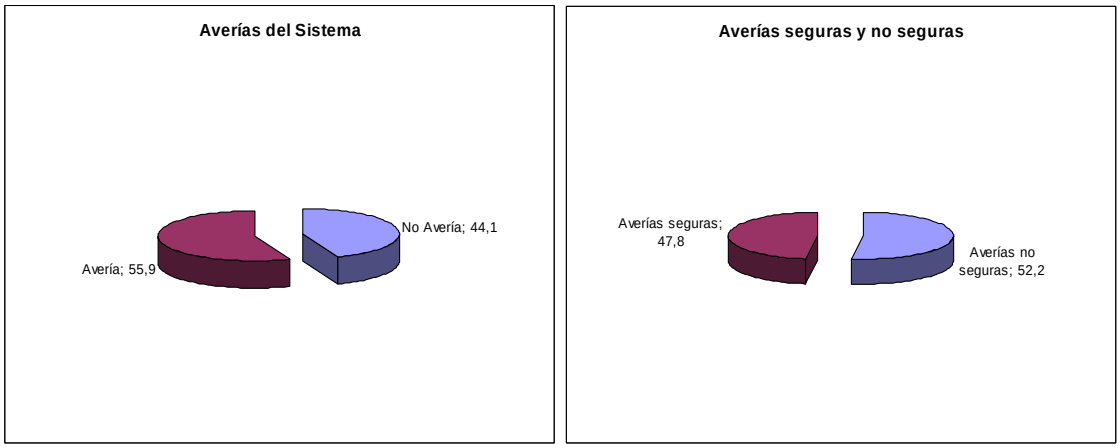


Figura 7.13: Averías del sistema seguras y no seguras

Por tanto, si se define la cobertura total del sistema a aquellas veces en las que éste, constituido por SO, microcontrolador y aplicación ha detectado que un error se ha producido, en forma de código de error o excepción, o por otro lado el sistema ha funcionado correctamente aún a pesar de los fallos inyectados, se tiene que la cobertura final del sistema ha sido:

$$C_{sist.compl.} = R0 + R1(\text{con detección}) + R2(\text{con detección}) + R3 = 733 + 0 + (525\text{-}82) + 57 = 1233\ (68{,}8\%)\ \text{casos}$$

Por tanto se puede decir que el 68,8% de las veces el sistema ha cubierto los errores introducidos en el mismo, bien produciendo un resultado correcto o bien detectando que un error se había producido en el sistema. En estos casos se puede decir que el funcionamiento del sistema podría ser seguro en este porcentaje, si la aplicación tratase correctamente los errores notificados antes de que éstos llegaran a convertirse en averías.

En cuanto a cobertura del sistema operativo se tiene que ésta ha sido como sigue:

$$C_{SO} = R0 + R1(\text{con detección}) + R2(\text{con detección}) = 733 + 0 + (525\text{-}82) = 1176\ (65{,}63\%)\ \text{casos}$$

En este caso la cobertura del SO vendrá dada por aquellos casos en los que el sistema finalizó la ejecución de la aplicación de un modo correcto, sumado a aquellos casos en los que el propio sistema operativo OSEK detectó un error a través de sus propios mecanismos de detección de errores. Por tanto se tiene que el resultado ha sido del 65,63% para la aplicación dada.

7.3.2 Códigos de error devueltos por el SO

En esta sección se va a estudiar cuales han sido los códigos de error devueltos por el SO durante los experimentos, su significado, porqué se han producido y qué efecto han tenido en el sistema.

Para el tratamiento de errores OSEK/VDX provee al usuario de una serie de rutinas específicas del sistema para el tratamiento del procesamiento interno del SO, como son los códigos de error. Estas rutinas hacen que el tratamiento de errores esté centralizado y además deja al usuario la libertad de diseñarlas para un tratamiento más especifico según sean los requerimientos del sistema. Lo que hace OSEK es proveer de un conjunto de rutinas que tienen prioridad sobre el resto de tareas del sistema, que son consideradas como parte del sistema operativo, y que tienen una interfaz estandarizada para diferentes implementaciones de la especificación OSEK/VDX.

Así para el tratamiento de los errores OSEK distingue dos tipos:

- **Errores de aplicación**: Ante este tipo de errores el SO no ejecuta el servicio solicitado, pero asume la corrección de sus datos internos. En este caso hay un tratamiento centralizado de errores dejando al diseñador del sistema definir el comportamiento del mismo a partir del error detectado.

- **Errores fatales (o críticos)**: En este caso el SO no puede asumir por más tiempo la corrección de sus datos internos y la estabilidad del sistema, y por tanto lo lleva a una situación segura de apagado del mismo.

En definitiva, como se puede observar que el tratamiento de errores en OSEK queda bastante definido y localizado, dando la suficiente libertad al diseñador del sistema para un tratamiento especifico que a los mismos se les quiera dar. A continuación se va a describir cuales han sido los resultados, entorno a los códigos de error obtenidos, tras la ejecución de los experimentos.

Tabla 7.14: Códigos de Error devueltos por el SO

		Frecuencia	Porcentaje	Porcentaje válido	Porcentaje acumulado
Válidos	Error 4101	197	44,7	44,7	44,7
	Error 4301	37	8,4	8,4	53,1
	Error 5301	78	17,7	17,7	70,7
	Error 5302	26	5,9	5,9	76,6
	Error 5501	88	20,0	20,0	96,6
	Error 6101	15	3,4	3,4	100,0
	Total	441	100,0	100,0	

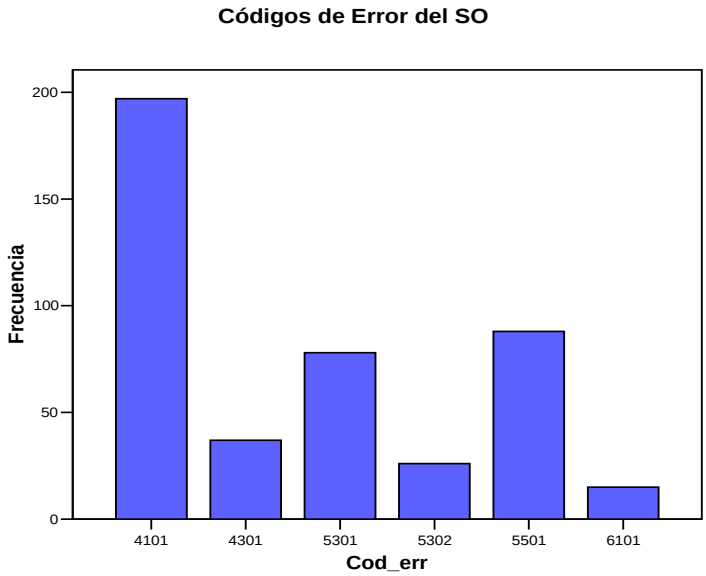


Figura 7.14: Frecuencias de los códigos de error devueltos por el SO

Clasificación y descripción de los códigos de error obtenidos:

- **Error 4101**: se ha obtenido cerca de un 45% de las veces, está codificado como osdErrSEWrongTaskID, y está dentro del grupo de códigos de error referentes al control de eventos. Se produce cuando se obtiene un identificador de tarea no válido. Este error ha producido un 37,7% de los cuelgues del sistema. Este código es devuelto por la llamada al sistema SetEvent cuando se intenta establecer un evento para un identificador de llamada no válido, en este caso la inyección ha afectado al parámetro que especificaba para qué tarea, se iba a activar un evento y se iba a pasar al estado de "Ready".

- **Error 4301**: se ha obtenido cerca de un 8,4% de las veces, está codificado como osdErrGEWrongTaskID, y también está dentro del grupo de códigos de error referentes al control de eventos. Es producido por la llamada al sistema GetEvent cuando no se puede obtener el estado de la máscara de los eventos pertenecientes a una tarea. Este error ha producido un 7% de los cuelgues del sistema, donde al no poder saber para una tarea que eventos se han cumplido se pierde la información referente a la situación en la que dicha tarea se encuentra.

- **Error 5301**: se ha obtenido cerca de un 17,7% de las veces, está codificado como osdErrSAWrongAlarmID, y está dentro del grupo de códigos de error referentes al manejo de alarmas. Es producido por la llamada al sistema SetRelAlarm cuando se obtiene un identificador de alarma no válido. Este error ha producido un 15% de los cuelgues del sistema donde al obtener un código de alarma inválido el sistema no puede

activar la tarea que estaba esperando tal evento y pierde la secuencia de acciones a realizar.

- **Error 5302**: se ha obtenido cerca de un 6% de las veces, está codificado como osdErrSAAlreadyActive, y como en el caso anterior también está dentro del grupo de códigos de error referentes al manejo de alarmas. Es producido por la llamada al sistema SetRelAlarm cuando se quiere activar una alarma que ya está activa y ya está asignada a otra tarea. Este error ha producido un 5% de los cuelgues del sistema, donde al intentar obtener una alarma perteneciente a otra tarea, la tarea que ha solicitado dicho servicio no puede ser activada y por tanto se pierde una vez más la secuencia de acciones a llevar a cabo.

- **Error 5501**: se ha obtenido cerca de un 20% de las veces, está codificado como osdErrCAWrongAlarmID, y como en el caso anterior también está dentro del grupo de códigos de error referentes al manejo de alarmas. Es producido por la llamada al sistema CancelAlarm cuando se quiere cancelar una alarma con un identificador no válido, es decir que no existe. Este error ha producido un 17% de los cuelgues.

- **Error 6101**: se ha obtenido cerca de un 3,5% de las veces, está codificado como osdErrSOStackOverflow. Este código está dentro del grupo de códigos de error referentes al control de la ejecución del propio SO. Es producido por la llamada al sistema osCheckStackOverflow cuando se realiza una comprobación de Overflow de la pila de la tarea que ha producido este error. Este código de error ha producido un 3% de los cuelgues totales del sistema.

La notificación de los códigos de error ha supuesto que el sistema pasara a un estado de cuelgue. Esto es debido a que ante la aparición de un error detectable no hay un tratamiento específico en el procesamiento de los códigos de error, dada la filosofía de OSEK/VDX, sino que tan sólo la notificación de los mismos, ya que la implementación de las acciones a realizar cuando un error es obtenido queda como tarea a realizar por los diseñadores o programadores del sistema, que en este caso no fue así.

A modo de conclusiones y tras la revisión de los códigos de error obtenidos, se puede concluir que la herramienta nos ofrece datos útiles acerca de dónde incidir más en la implementación de mecanismos de tolerancia a fallos para una aplicación determinada. En el caso que nos ocupa, para la aplicación de control de semáforos, se ha podido observar como la mayoría de los códigos de error obtenidos vienen por parte de las llamadas al sistema que manejan alarmas y eventos, donde la obtención de identificadores no válidos hacen que el sistema pueda perder la secuencia de acciones a realizar. Esto lleva a que el sistema entre en un estado impredecible sin que se sepa qué está ocurriendo en el interior del SO, visto como esa caja negra a la que se le solicitan unos servicios determinados a través de su API. Por tanto, conociendo cuales son los códigos de error más típicos del sistema operativo, que para una aplicación determinada se pueden obtener, se puede saber cuales van a ser las llamadas al sistema más críticas. Esto permitiría al diseñador del sistema poder establecer las políticas y mecanismos de prevención de fallos necesarios para una aplicación determinada.

7.3.3 Excepciones

Una vez se han estudiado los códigos de error obtenidos por parte del SO, a continuación se va a analizar cuáles han sido las excepciones lanzadas por el microcontrolador MPC565, que se corresponde con aquellos casos en los que tras la inyección de un fallo, los mecanismos de detección de errores del microcontrolador son los que detectan un error en el sistema y en este

caso lo llevan a finalizar la ejecución. Estos han sido codificados como salida “R3” del experimento y tan sólo han supuesto un 3,2% del total de los experimentos.

En la siguiente gráfica de la figura 7.15 se puede observar qué excepciones se han obtenido y en qué porcentaje:

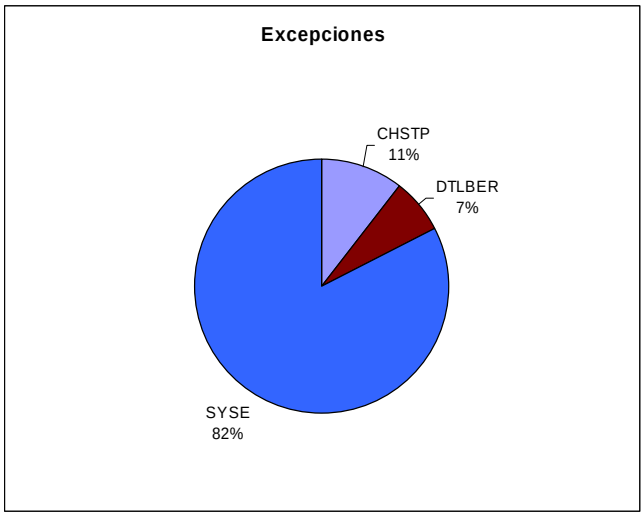


Figura 7.15: Frecuencia de obtención de las Excepciones

En primer lugar hay que decir que la tasa de experimentos que han finalizado con el lanzamiento de una excepción por parte del microcontrolador, para la carga de trabajo de este caso, han sido muy pocos. De entre estos experimentos se observa que la excepción que con mayor frecuencia se ha obtenido es la de “excepción de llamada al sistema” (SYSE) aunque tan sólo en 47 casos. Estas se lanzan a causa de una instrucción de llamada al sistema errónea. En segundo lugar se tiene la excepción de “*checkstop*” (CHSTP) que se produce tras una excepción de “*machine check*”, con lo cual una es consecuencia de la otra, tras la cual se produce una suspensión en el procesamiento de instrucciones y el sistema debe ser reseteado para ser recuperado. En estos casos esto puede ser debido por al acceso a una dirección que no existe, o un error en los datos o una violación de protección de almacenamiento. Y en último lugar y en tan sólo 4 ocasiones, la excepción de “errores de protección de datos” (DTLBER) donde se agrupan errores producidos por el acceso a memoria protegida.

7.3.4 Tiempos

En el siguiente apartado se va estudiar cómo las campañas de inyección de fallos han podido incidir en los resultados temporales que se esperan de un funcionamiento correcto del sistema. Como se ha adelantado en los primeros capítulos, las averías del sistema se pueden producir tanto por la obtención de valores incorrectos durante la ejecución del sistema, y por tanto se habla de averías en el dominio del valor, como por averías en el dominio del tiempo, por la obtención de resultados fuera de los límites temporales marcados como “*deadlines*” para la obtención de dichos valores correctos. Por tanto, a continuación se va a examinar los tiempos de detección de errores del SO, los tiempos de las tareas de la aplicación en ejecución libre de errores, y la influencia de los fallos inyectados en los tiempos de ejecución de las mismas.

7.3.4.1 Tiempos de detección del SO

En este apartado se van a calcular los tiempos de detección de errores del SO, siguiendo la metodología anteriormente propuesta para la obtención de tiempos de latencia de detección.

Para ello se obtiene la diferencia temporal entre el instante en el que el fallo se hace efectivo (se activa) en el sistema y el instante en el cual el mecanismo de detección de errores del SO en cuestión notifica un código de error indicando un estado erróneo del sistema.

A continuación, se va estudiar cual es el coste temporal de detectar cada unos de los códigos de error producidos por el sistema operativo (valores en microsegundos):

Tabla 7.15: Tiempos de detección de errores del SO

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Cod_Err_4101	196	28,24	28,26	28,2506	,00071	,01001	,000
N válido (según lista)	196						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Cod_Err_4301	36	28,58	40,66	32,5700	,94218	5,65308	31,957
N válido (según lista)	36						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Cod_Err_5301	77	30,10	30,10	30,1000	,00000	,00000	,000
N válido (según lista)	77						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Cod_Err_5302	25	40,34	63,94	44,8184	1,82672	9,13358	83,422
N válido (según lista)	25						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Cod_Err_5501	87	27,78	27,82	27,8009	,00056	,00520	,000
N válido (según lista)	87						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Cod_Err_6101	15	286,00	310393,00	82859,1707	26044,517	100869,98	1,02E+10
N válido (según lista)	15						

En la tabla 7.15 se pueden observar los tiempos medios, valores máximos y mínimos obtenidos con el objetivo de calcular, cuál es el coste temporal que supone tener una notificación, por parte del SO, de que un error ha sido detectado durante la ejecución de la aplicación. De los estadísticos anteriores destacar que en este caso, y para la aplicación dada, los tiempos de detección de errores relacionados con el control de eventos de las tareas están entorno a los 28,25us. Este dato es importante ya que aporta información interesante para poder acotar el tiempo necesario para recuperar al sistema de una situación habitualmente de cuelgue. En su mayor parte, esto es debido a que si la información de las estructuras que almacenan el control de eventos para las tareas se corrompe, el sistema puede perder la secuencia de las siguientes acciones a ser llevadas a cabo.

Con respecto a los errores relacionados con el control de eventos temporizados, como es el control de alarmas muy utilizado en OSEK para la sincronización de las tareas, dichos errores se han sido detectado entre los [27,8ms, 44,8ms]. Y por último, con respecto a los errores de desbordamiento de la pila se observa que en este caso la variabilidad en el tiempo de detección es muy alta. En este tipo de errores los valores han abarcado tiempos que van desde los 286us a los 310ms, lo cual hace que estimar el tiempo medio de detección de este tipo de errores sea muy difícil. Por suerte sólo se han producido en un 3,5% de las veces en las que se ha detectado errores.

A continuación y de un modo no tan específico se van a mostrar los datos totales sobre los tiempos de detección de errores del SO en todos los casos, exceptuando el caso de los errores de *overflow* de pila que disparan los tiempos. Así por tanto se tiene que (valores en microsegundos):

Estadísticos

N	Válidos	421
	Perdidos	0
Media		29,8491
Error típico de la media		,23614
Mediana		28,2600
Moda		28,26
Desviación típica		4,845188
Varianza		23,476
Mínimo		27,78
Máximo		63,94

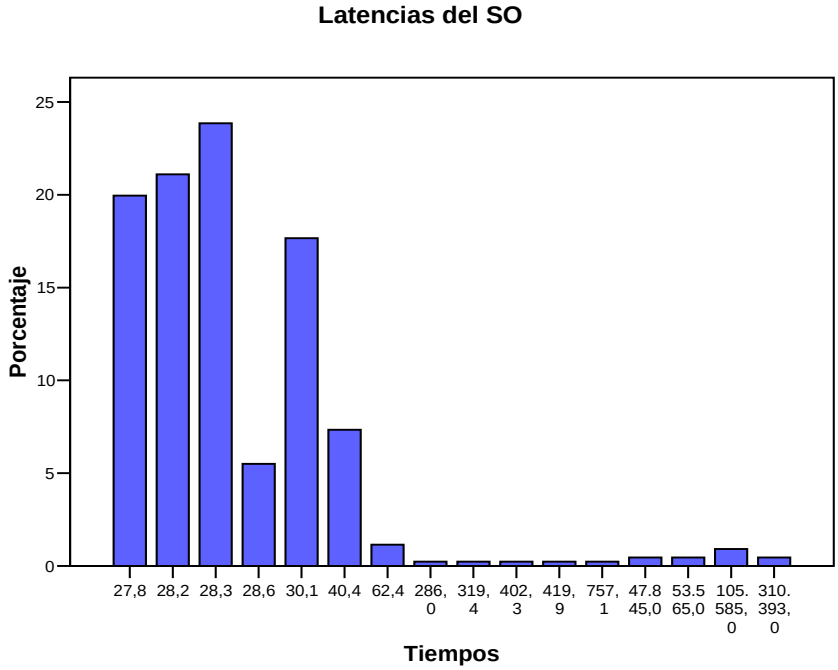


Figura 7.16: Frecuencia de las latencias de detección

Como se puede observar los tiempos de detección de errores del SO, por supuesto para la aplicación dada, están entorno a una media de 29,85us, ya que como se puede extraer de esta última gráfica de la figura 7.16, en más del 85% de casos las latencias has oscilado entre los 27,82us y los 30,1us. Por tanto viendo la distribución de tiempos que muestra la figura 7.16, se puede afirmar que las latencias de detección de errores se centran más en dicha franja de

tiempos, al ser éstos los tiempos de latencia que con mayor frecuencia se han obtenido y es por ello que la mediana y la moda se corresponden con 28,26us.

7.3.4.2 Tiempos de las Tareas

A continuación en este apartado se va mostrar cual ha sido el efecto de los fallos inyectados con respecto a los tiempos de ejecución de las tareas. El objetivo es poder observar cómo la inyección de fallos de diseño del software, en un sistema constituido por componentes COTS, puede afectar a los propios componentes en el cumplimiento del trabajo asignado a cada una de las tareas del sistema dentro de los tiempos establecidos. Con esto se puede ver que a pesar de que los valores obtenidos al final son correctos, los tiempos en los que estos valores son producidos pueden variar. Estas variaciones temporales que se mencionan podrán ser más o menos críticas dependiendo del criterio del diseñador de la aplicación, a la hora de establecer los tiempos máximos asignados a las tareas, y que delimitarían las averías del sistema en el dominio del tiempo. En la siguiente tabla 7.16, se pueden ver las diferentes tareas que se ejecutan en el sistema, su función y el nombre que se les ha dado a la hora de identificar los tiempos obtenidos.

Tabla 7.16: Descripción de las tareas del sistema.

Tarea	Descripción
Control	Bucle de control de la gestión del tráfico para cada calle
SensorA	Bucle de control de la llegada/salida de vehículos en el semáforo A
SensorB	Bucle de control de la llegada/salida de vehículos en el semáforo B
SensorC	Bucle de control de la llegada/salida de vehículos en el semáforo C
SensorD	Bucle de control de la llegada/salida de vehículos en el semáforo D
TrafficIn	Bucle que simula la llegada de vehículos a cada una de las calles
TrafficOut	Bucle que simula la salida de vehículos de cada una de las calles

Los siguientes estadísticos representan en primer lugar los datos obtenidos correspondientes a los tiempos máximos de ejecución de las tareas para aquellos experimentos en los que no se inyectaban fallos (ejecuciones libres de fallos). En estos casos, dado que los resultados de los experimentos son deterministas se han tomado un conjunto muestras suficiente para establecer la referencia temporal de las tareas en ejecución libre de fallos. Seguidamente se muestran los resultados obtenidos para los tiempos máximos de ejecución de las tareas de la aplicación, pero una vez se han inyectado fallos en el sistema. En estos casos los tiempos que se obtienen se corresponden con aquellos experimentos que se han codificado como “R0”, donde se observaba que la inyección de un fallo finalmente no afectaba, desde el punto de vista del dominio del valor, a la entrega del servicio. Así por tanto, lo tiempos obtenidos tarea por tarea son los siguientes (valores en milisegundos):

Tabla 7.17: Tiempos para la tarea Control sin fallos y con fallos

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Control	116	,218	2,840	1,99867	,111456	1,200414	1,441
N válido (según lista)	116						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Control	472	,043	5,232	1,92830	,061078	1,326948	1,761
N válido (según lista)	472						

Como se ha mencionado, en primer lugar se presentan los datos correspondientes a las ejecuciones libres de errores y en segundo lugar los datos correspondientes a los experimentos donde se han inyectado fallos. En primer lugar se tiene la tarea Control. Para esta tarea se puede observar que la media para los tiempos máximos en presencia de fallos se reduce en unos 70us, lo cual indica que en los experimentos donde se han inyectado fallos, ha habido casos en los cuales la tarea ha finalizado antes de lo esperado. Por tanto se podrían haber producido averías por entrega temprana del servicio. Como se puede ver en los estadísticos de la tabla 7.17 los tiempos mínimos de la tarea han sido de 43us, que supone haber acabado 175us antes de lo esperado en el caso del valor mínimo. En el caso de los valores máximos se tiene que la tarea se puede haber retardado hasta los 5,23ms, lo que supone finalizar 2,39ms más tarde de lo esperado, posiblemente provocando en este caso averías por entrega tardía del servicio.

Tabla 7.18: Tiempos para la tarea SensorA sin fallos y con fallos

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
SensorA	116	,238	2,862	2,63205	,060930	,656239	,431
N válido (según lista)	116						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
SensorA	472	,137	52,400	2,70933	,121514	2,639959	6,969
N válido (según lista)	472						

Para la tarea SensorA se observa que la media en presencia de fallos se incrementa en 77us. Aunque lo más significativo es que en este caso los tiempos máximos pueden llegar a alcanzar los 52,4ms, provocando claramente, los casos en los que se hayan producido estos tiempos, averías por un retardo excesivo en la entrega del servicio.

Tabla 7.19: Tiempos para la tarea SensorB sin fallos y con fallos

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
SensorB	116	,238	2,859	1,37321	,117470	1,265193	1,601
N válido (según lista)	116						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
SensorB	472	,042	21,932	2,11885	,099561	2,163019	4,679
N válido (según lista)	472						

Para la tarea SensorB se tiene que la media en presencia de fallos se incrementa en 0,745ms. Para esta tarea se han obtenido tanto tiempos mínimos como máximos que distan bastante de sus valores normales, llegando a alcanzar en el peor de los casos un tiempo máximo de 21,932ms, y con respecto al valor mínimo de 42us. Por lo que se puede concluir que los fallos inyectados han hecho que la tarea en ocasiones haya producido tanto averías por retardo como por adelanto en lo que sería la entrega del servicio.

Tabla 7.20: Tiempos para la tarea SensorC sin fallos y con fallos

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
SensorC	116	,241	2,869	2,14062	,104413	1,124559	1,265
N válido (según lista)	116						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
SensorC	472	,042	22,383	1,92966	,093088	2,022381	4,090
N válido (según lista)	472						

Para la tarea SensorC se observa que en este caso la media en presencia de fallos se reduce en 211us con respecto a los experimentos sin fallos. Para esta tarea se han obtenido tanto tiempos mínimos como máximos similares al anterior caso correspondiente al SensorB, por lo que se obtienen conclusiones similares. Aunque posteriormente en las gráficas de la distribución de los tiempos de ejecución, se podrá ver si ambas tareas se han visto o no muy perjudicadas en sus tiempos de ejecución por efecto de los fallos inyectados.

Tabla 7.21: Tiempos para la tarea SensorD sin fallos y con fallos

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
SensorD	116	,238	2,948	2,06203	,104977	1,130634	1,278
N válido (según lista)	116						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
SensorD	472	,042	53,599	2,13697	,225262	4,893942	23,951
N válido (según lista)	472						

Para la tarea SensorD se tiene que en este caso la media en presencia de fallos se incrementa en 74us. Aunque lo más significativo es que para esta tarea los tiempos máximos, como en el caso del SensorA, se disparan hasta los 53,59ms, cuando lo esperado es como mucho una respuesta por parte de dicha tarea a los 2,94ms. En la distribución de los tiempos también se podrá observar si esto se ha producido en muchas ocasiones. En cuanto a los tiempos mínimos se puede ver que la tarea puede ofrecer una respuesta a los 42us, cuando como mínimo esto debería suceder a los 238us.

Tabla 7.22: Tiempos para la tarea TrafficIn sin fallos y con fallos

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
TrafficIn	116	2,713	2,872	2,81541	,004584	,049369	,002
N válido (según lista)	116						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
TrafficIn	472	,042	4,147	2,72307	,022303	,484538	,235
N válido (según lista)	472						

Para el caso de la tarea TrafficIn de los estadísticos de la tabla 7.22 se puede extraer que en este caso la media en presencia de fallos se reduce en 0,745ms. Esto supondría incrementar la

llegada de vehículos al cruce antes de lo esperado. Además, se observa que acorde a esto el tiempo mínimo de respuesta es de 42us, cuando lo esperado es como mínimo 2,71ms. Y en cuanto a los tiempos máximos, pues también se aprecia que estos pueden llegar a duplicarse, reduciendo en este caso por el contrario el número de vehículos que llegan al cruce.

Tabla 7.23: Tiempos para la tarea TrafficOut sin fallos y con fallos

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
TrafficOut	116	2,770	2,772	2,77134	,000066	,000711	,000
N válido (según lista)	116						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
TrafficOut	472	,108	2,844	2,60459	,029568	,642387	,413
N válido (según lista)	472						

En última instancia para la tarea TrafficOut se tiene que en este caso la media en presencia de fallos se reduce en 167us con respecto a la media de los experimentos libres de fallos. Aunque lo más significativo es que los tiempos mínimos se van hasta los 108us cuando lo esperado es de 2,77ms, esto significaría que en algún experimento algún semáforo estaría dando paso a vehículos antes de tiempo.

7.3.4.3 Distribución de los tiempos máximos

En al apartado anterior se han estudiado los valores obtenidos de los estadísticos producidos a partir de la evaluación de los tiempos máximos de ejecución de las tareas. A continuación se va analizar, cuál es la distribución de esos tiempos máximos para cada una de las tareas en los experimentos que se han llevado a cabo. El objetivo es poder observar cual de dichas tareas se ha visto mayormente afectada en sus tiempos de ejecución a causa de los fallos introducidos y cuales de éstas serían más propensas a introducir retardos o adelantos en el tiempo total de ejecución del sistema.

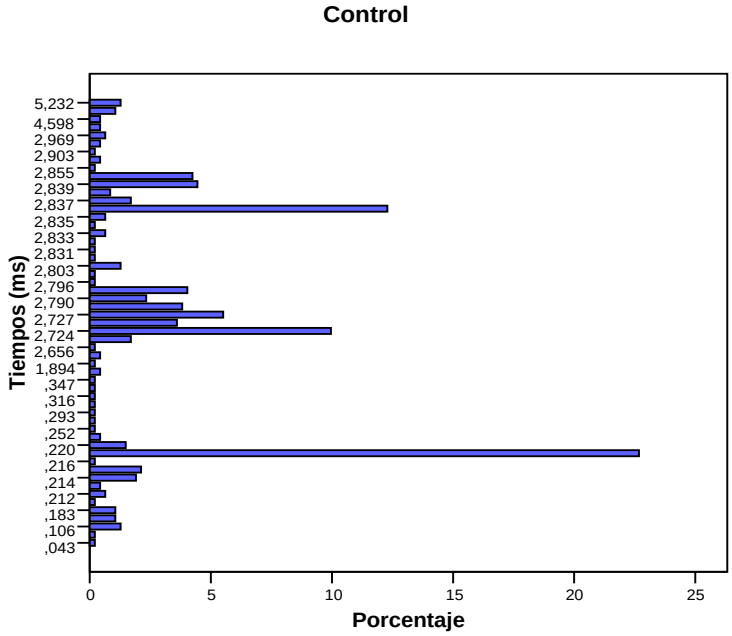


Figura 7.17: Distribución de los tiempos máximos para la tarea Control

En primer lugar para la tarea Control como se puede ver en la figura 7.17, se observa que la mayoría de los tiempos de ejecución están por encima de los 2ms, hay que recordar que la media en condiciones normales era de 1,998ms. A partir de los 2,840ms, que era el valor tope, también hay entorno a un 8% de experimentos donde se habría superado este tiempo máximo y que se corresponderían con averías por retardo en la entrega del servicio. Con respecto a los tiempos que están por debajo de los 218us también se puede observar, en una proporción aproximada a las averías por retardo, que se han producido averías por adelanto.

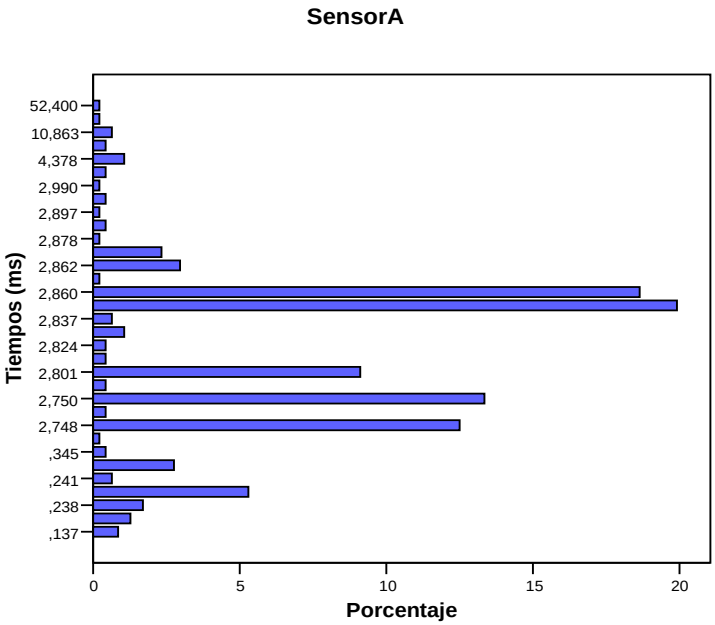


Figura 7.18: Distribución de los tiempos máximos para la tarea SensorA

Con respecto a la tarea SensorA, según la figura 7.18 se puede observar que sí se tiene en cuenta que los rangos de tiempos para esta tarea estaban entorno a [0,238ms, 2,862ms], se aprecia que en este caso no hay muchos valores que estén por debajo del valor mínimo; no sucediendo así por el contrario con los valores máximos, donde sí se puede ver, que se han obtenido entorno a un 10% de valores superiores al valor máximo y que se corresponderían con averías por retardo.

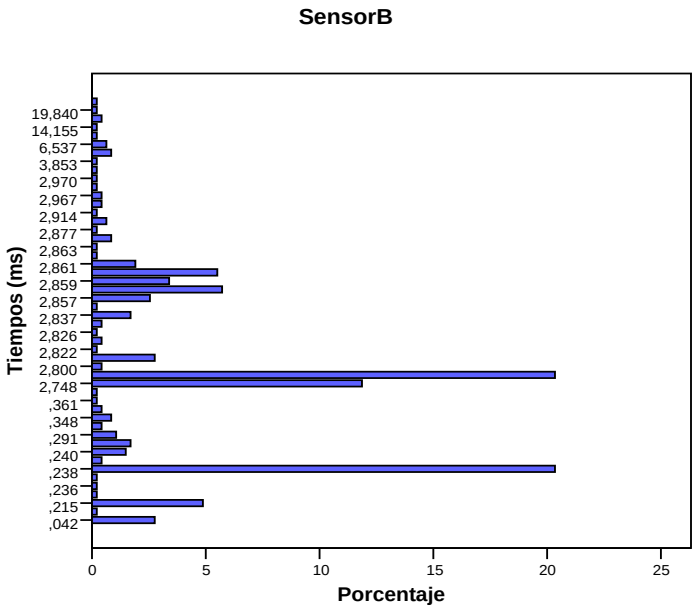


Figura 7.19: Distribución de los tiempos máximos para la tarea SensorB

Con respecto a la tarea que implementa el SensorB, en la figura 7.19 se puede ver que la mayoría de los tiempos se hallan a partir de los 2,748ms. Se observa que la tarea pasa de los 361us a los 2,748ms, estando la media en 1,373ms, donde en la gráfica se puede ver que no existe ningún experimento para dicha media. En este caso se tiene, que la tarea o se ejecuta entorno a los 238us o ya pasa a tardar en ejecutarse alrededor de los 2,8ms. A partir de los 2,859ms, que es el valor máximo en ausencia de fallos, se aprecia que también hay un porcentaje elevado de tiempos que están por encima provocando averías por retardo.

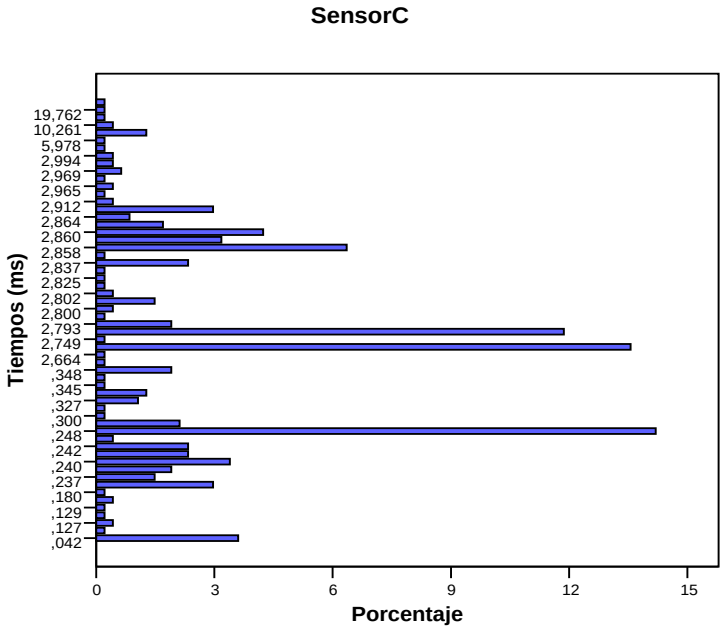


Figura 7.20: Distribución de los tiempos máximos para la tarea SensorC

Con respecto a la tarea correspondiente al SensorC, el rango de tiempos para ésta se encuentra en el intervalo [0,241ms, 2,869ms]. Según la figura 7.20 se puede observar que por debajo de los 241us hay alrededor de un 5% de experimentos donde la tarea ha finalizado antes de lo esperado. Por encima de los 2,869ms también hay cerca de un 7% de experimentos donde la entrega del servicio está fuera de plazo.

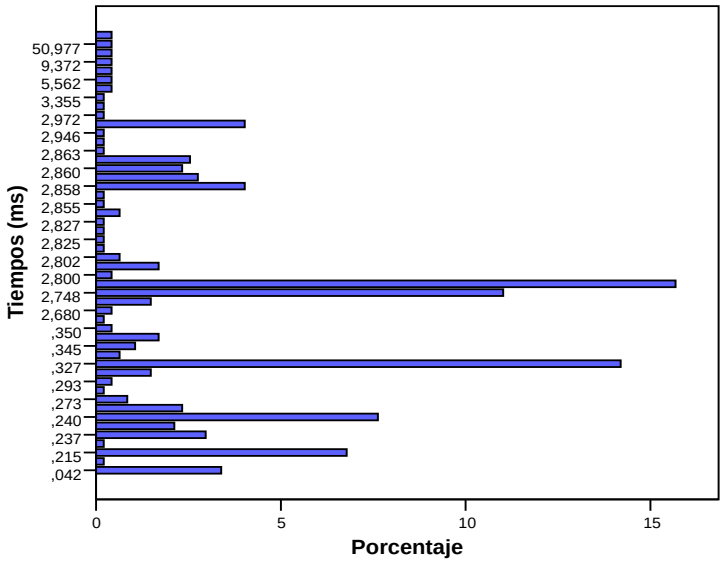


Figura 7.21: Distribución de los tiempos máximos para la tarea SensorD

Con respecto a la tarea SensorD, los tiempos para ésta estaban en el intervalo [0,238ms, 2,948ms]. Como en casos anteriores hay valores que se salen del rango tanto por encima como por debajo de los valores límite, aunque en este caso hay un porcentaje cercano al 11% de averías por adelanto en la entrega del servicio. Por encima de los 2,948ms remarcar los 50,977ms a los que ha llegado la tarea en varios experimentos, donde a pesar de ello los valores obtenidos fueron correctos, aunque evidentemente desde el punto de vista temporal no.

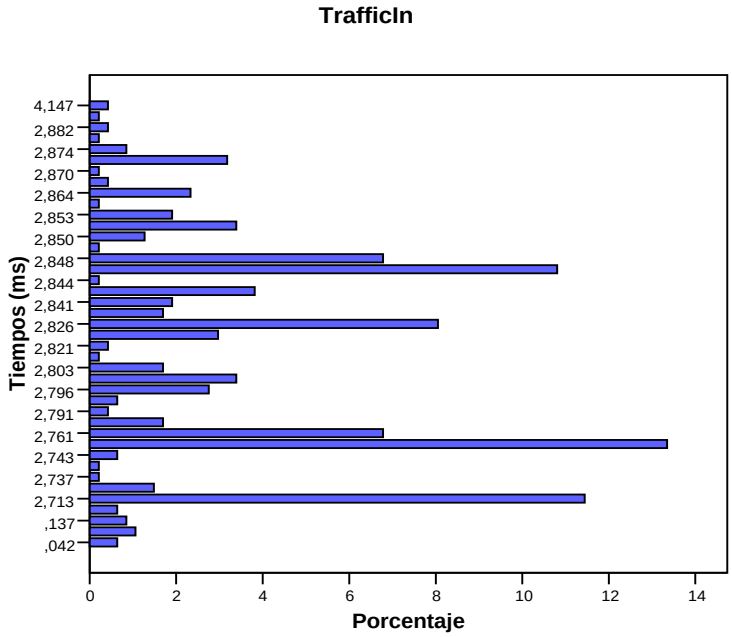


Figura 7.22: Distribución de los tiempos máximos para la tarea TrafficIn

Con respecto a la tarea que implementa TrafficIn si se tiene en cuenta que los tiempos para ésta estaban entorno a [2,713ms, 2,872ms], según la gráfica 7.22 se puede observar que dicha tarea no se ha visto muy afectada por los fallos inyectados, ya que la mayoría de los tiempos se encuentran dentro del intervalo que se obtuvo para los experimentos libres de fallos. Como se puede observar hay muy pocos casos que se salen fuera del rango.

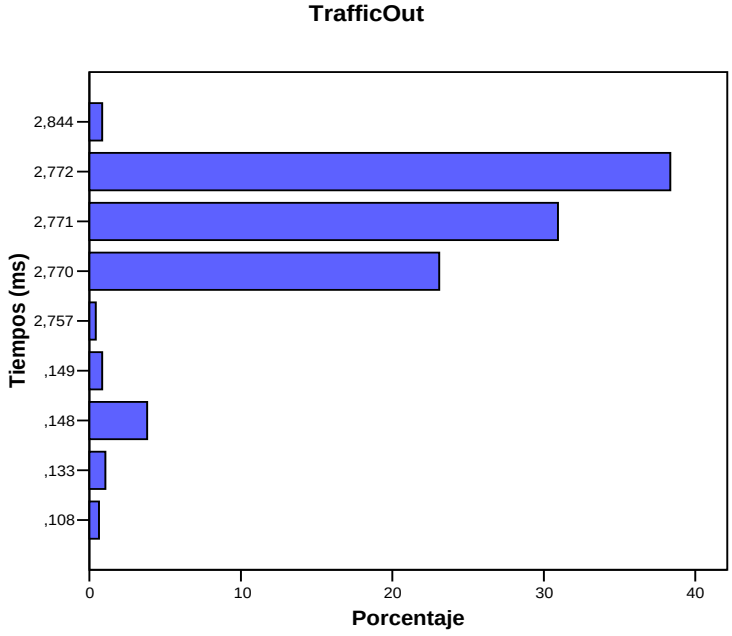


Figura 7.23: Distribución de los tiempos máximos para la tarea TrafficOut

Finalmente con respecto a la tarea que implementa TrafficOut, se observa en primer lugar que la variabilidad de los tiempos obtenidos no es tan acusada como con el resto de tareas del sistema. También se puede apreciar que la mayoría de los tiempos, entorno a un 93%, se encuentra alrededor de los 2,771ms. Teniendo en cuenta que TrafficOut pone los semáforos en verde, aquellos valores de tiempos de ejecución que están por encima de los 2,772ms, que son más bien pocos, no serían tan preocupantes tanto como sí lo serían los valores que están por debajo de los 2,770ms, donde se habrían producido averías por adelanto en la entrega del servicio provocando que algún semáforo cambiara de color antes de tiempo, y que en este caso han sido más numerosos.

7.4 RESULTADOS PARA MICROC/OS-II Y EL CONTROL DE SEMÁFOROS

En este último apartado de resultados se van a estudiar los datos obtenidos para un sistema que incorpora una carga constituida por el sistema operativo de tiempo real MicroC/OS-II y una aplicación de control de semáforos, como en el caso anterior. El objetivo es poder comparar cual de los dos sistemas operativos mejora la confiabilidad final del sistema para una aplicación determinada. En este caso, se han realizado 3000 experimentos y el análisis llevado a cabo sobre los resultados obtenidos sigue la misma estructura que en los casos anteriores, dando así al lector la oportunidad, como se comentó anteriormente, de poder recurrir de forma independiente a ver datos referentes a una carga determinada.

7.4.1 Coberturas

En primer lugar se va a estudiar cuales han sido los resultados que se obtienen en cuanto a coberturas de detección tanto del propio sistema operativo bajo estudio, en este caso MicroC/OS-II, como sobre la cobertura final del sistema. Para ello habrá que obtener, además de los datos del SO, datos relativos al propio microcontrolador MPC565 y a la aplicación que simula el control de semáforos. El objetivo es ver si tras la inyección de un fallo de diseño del software en el sistema, éste se traduce en un error detectado o no y finalmente en una avería en lo que sería la entrega final del servicio.

Así primeramente, se va a definir cómo se ha codificado el resultado final de cada experimento. Esta codificación que se va a explicar no hace referencia a la cobertura final del sistema, sino que detalla con respecto a la aplicación, como finaliza el experimento. Para obtener datos sobre coberturas habrá que tener en cuenta la detección de errores y otros detalles que más adelante se van a calcular.

De las gráficas siguientes y teniendo en cuenta las posibles salidas al experimento, como se describió anteriormente en el capítulo 6, los resultados obtenidos son los siguientes:

Estadísticos

N	Válidos	3000
	Perdidos	0

Tabla 7.24: Resultados de la ejecución

		Frecuencia	Porcentaje	Porcentaje válido	Porcentaje acumulado
Válidos	R0	912	30,4	30,4	30,4
	R1	946	31,5	31,5	61,9
	R2	89	3,0	3,0	64,9
	R3	1053	35,1	35,1	100,0
	Total	3000	100,0	100,0	

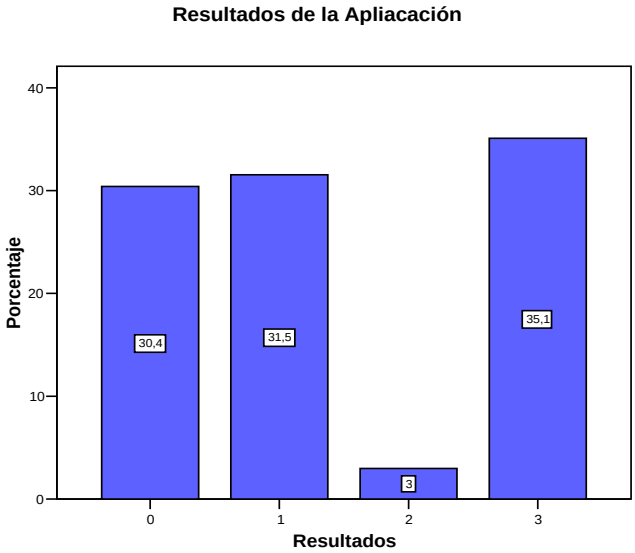


Figura 7.24: Resultados de la ejecución

La consecución del experimento se ha codificado de la siguiente manera:

- **R0**: No hubo error en la entrega del servicio.
- **R1**: El servicio entregado no fue el esperado.
- **R2**: El sistema quedó en una situación de ejecución sin respuesta al exterior (cuelgue del sistema). La aplicación no produce ningún valor de salida.
- **R3**: El microcontrolador produjo una excepción.

Acorde a esto y tal y como se puede apreciar en la gráfica y tabla de resultados de la figura y tabla 7.24, se observa que en un 30,4% de las veces se ha obtenido un resultado de "R0", esto se corresponde con aquellos experimentos en los que aún a pesar de haberse producido la inyección efectiva de un fallo, la ejecución de la aplicación y por tanto la entrega final del servicio ofrecido por el sistema no se vieron afectados por la inyección de dicho fallo. En estos casos se puede intuir que los mecanismos de tolerancia a fallos del sistema podrían haber actuado o que la inyección podría haberse realizado en localizaciones de memoria que finalmente no son utilizadas por el sistema operativo o arquitectura del microcontrolador, como podrían ser partes altas de palabra de los parámetros de las llamadas, o bits no significativos de los parámetros, o también en ocasiones en parámetros que albergan datos devueltos por las llamadas y que por tanto son reescritos. Por tanto en estos casos se puede concluir que la inyección de fallos no afectó al sistema y la ejecución de la aplicación finalizó de un modo correcto.

Cuando se obtuvo un resultado de "R1", que fue en un 31,5% de las veces, en estos casos la inyección del fallo tan sólo afectó a la entrega del servicio, ya que la aplicación se ejecutó normalmente sin excepciones ni cuelgues aunque los resultados obtenidos no fueron correctos, o

al menos los esperados, lo cual finalmente supone una avería del sistema. En estos casos haría falta disponer de algún oráculo [Dbench04] que indicara que los valores de salida que se obtienen por parte de la aplicación no son correctos o no están dentro de los rangos permitidos, y así poder detectar este tipo de averías. En el caso que nos ocupa, esto se implementa a través de una doble ejecución con y sin fallos para determinar cuáles son los valores correctos de la ejecución. Esta situación, que se ha denominado como "R1" es bastante crítica, ya que en estos casos parece ser que el sistema está funcionando correctamente pero los resultados que ofrece son incorrectos.

En estos casos para obtener la cobertura final del sistema se tiene que ver si durante esa supuesta correcta ejecución se ha producido alguna notificación por parte del sistema operativo a modo de código de error, que notifique a la aplicación que algo erróneo ha sucedido. Como se verá posteriormente, si hubiera habido casos en los que el SO hubiera producido algún código de error entonces se hubiera podido concluir que el error había sido detectado. En caso de que no se produzca ninguna notificación se puede hablar de que el sistema se halla ante un fallo crítico del mismo, ya que ninguno de los mecanismos del sistema operativo, ni del microcontrolador, puesto que estos casos se corresponden con los que no se producen tampoco excepciones, han detectado la avería del sistema.

En los casos en los que se obtuvo un resultado codificado como "R2", fueron en aquellas situaciones en las cuales el sistema no estaba ofreciendo claramente ningún servicio, son aquellas ocasiones en las cuales el sistema se halla en una situación de cuelgue, y esto se produjo en un 3% de las veces. Esta situación, que bien podría englobarse dentro de la anterior, puesto que el servicio final que se está ofreciendo es defectuoso o erróneo, ya que como se ha dicho no hay servicio, se ha querido distinguirla del resto porque a diferencia de la anterior, en donde el sistema se encontraba en una situación donde parecía ser que éste funcionaba bien; aquí claramente el sistema no funciona y además no se sabe lo que está haciendo, lo cual supone que se está también ante una avería del sistema. Como en el caso anterior, para obtener la cobertura final del sistema completo y del SO, se tendrá que observar si en esa situación se ha producido algún código de error por parte del SO que permitiera al diseñador de la aplicación recuperar al sistema.

Por último los casos en los cuales se ha obtenido un resultado de "R3", entorno a un 31,5% de las veces, han sido en aquellas ocasiones en las cuales el microcontrolador, con sus mecanismos internos de detección, ha sido el que ha detectado el error y lo ha notificado con el lanzamiento de una excepción y consiguiente parada de la ejecución del sistema. En este caso es el hardware el que ha sido el primero en detectar dicho error.

A continuación en la siguiente tabla 7.25, y con el fin de obtener datos para hallar la cobertura final del sistema y la del propio SO, se va a ver cuantos y cuales han sido los códigos de error (errores detectados) producidos por el SO en cada uno de los experimentos.

Tabla 7.25: Tabla de contingencia Resultado * Código de Error del SO

		Códigos de Error						
		0 (No error)	Error 1	Error 10	Error 20	Error 40	Error 42	Total
Resultado	R0	912	0	0	0	0	0	912
	R1	935	11	0	0	0	0	946
	R2	37	22	1	17	3	9	89
	R3	1025	0	0	0	28	0	1053
Total		2909	33	1	17	31	9	3000

Como se puede observar en la tabla 7.25, se tiene que se han obtenido 912 (30,4%) casos en los cuales ni se ha producido una avería, la ejecución es correcta, ni se ha obtenido ningún código de error. De entre los 946 experimentos en los cuales se ha obtenido un resultado

codificado como “R1”, es decir una avería del sistema, en este caso se han producido 11 (1,16%) casos donde se notificó un código de error por parte del SO, éstos se corresponden con averías seguras. Y por otro lado, 935 (98,8%) casos correspondientes también a “R1”, en los que no se ha producido ningún código de error y por tanto se puede hablar de casos de averías no seguras.

En el supuesto de los resultados codificados como “R2”, se ha obtenido que de entre un total de los 89 experimentos en los que se produjo un cuelgue del sistema, en 52 (58,42%) casos se detectó un error por parte del SO y por tanto también se habla de casos de averías seguras. Por el contrario, en 37 (41,57%) casos de cuelgues del sistema no hubo detección por parte del SO, correspondiéndose en éstos con averías no seguras.

Por último, en los casos en los que se produjo una excepción en el sistema que fue en 28 (2,66%) de ellos, antes del lanzamiento de la excepción además se produjo una notificación de error por parte del SO.

Así, se puede concluir que de las averías totales del sistema (R1+R2), que son un total de 1035 (34,5%) casos, en 63 (6%) casos hubo detección de errores, por tanto averías seguras y en 972 (94%) casos no la hubo, por tanto averías no seguras. Hay que precisar que en el caso de las averías denominadas como seguras, la detección de errores ofrecería, en caso de que así hubiera sido tratado, la posibilidad de que el sistema sea llevado a un estado seguro o a una recuperación del servicio. Así según se puede ver en las gráficas siguientes de la figura 7.25, el porcentaje de averías seguras y no seguras ha quedado como sigue:

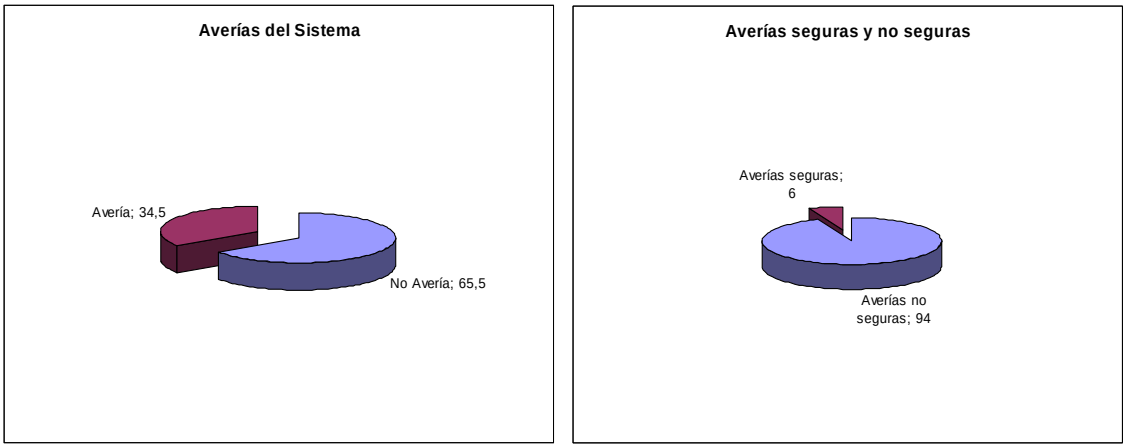


Figura 7.25: Averías del sistema seguras y no seguras

Por tanto, si se define la cobertura total del sistema como aquellas veces en las que éste, constituido por SO, microcontrolador y aplicación ha detectado que un error se ha producido, en forma de código de error o excepción, o por otro lado el sistema ha funcionado correctamente aún a pesar de los fallos inyectados, se tiene que la cobertura final del sistema ha sido:

$$C_{sist.compl.} = R0 + R1(\text{con detección}) + R2(\text{con detección}) + R3 = 912 + 11 + 52 + 1053 = 2028\ (67{,}6\%)\ \text{casos}$$

Por tanto se puede decir que el 67,6% de las veces el sistema ha cubierto los errores introducidos en el mismo, bien produciendo un resultado correcto o bien detectando que un error se había producido en el sistema. En estos casos se puede decir que el funcionamiento del sistema podría ser seguro en este porcentaje, si la aplicación tratase correctamente los errores notificados antes de que estos llegaran a convertirse en averías.

En cuanto a cobertura del sistema operativo se tiene que ésta ha sido como sigue:

$$C_{SO} = R0 + R1(\text{con detección}) + R2(\text{con detección}) = 912 + 11 + 52$$
$$= 975\ (32{,}5\ \%)\ \text{casos}$$

En este caso la cobertura del SO vendrá dada por aquellos casos en los que el sistema finalizó la ejecución de la aplicación de un modo correcto, sumado a aquellos casos en los que el propio MicroC/OS-II detectó un error a través de sus propios mecanismos de detección de errores. Por tanto se tiene que el resultado ha sido del 32,5% para la aplicación que se ha evaluado.

7.4.2 Códigos de error devueltos por el SO

En esta sección se describe cuáles han sido los códigos de error devueltos por el SO y su significado, porqué se han producido y qué efecto han tenido en el sistema.

En este caso, para el tratamiento de errores, MicroC/OS-II verifica en las propias llamadas al sistema operativo si se ha producido alguna situación anómala que deba ser tratada con especial atención, o si el servicio que se pide no puede ser ofrecido. En tal caso, la llamada al sistema operativo devolverá un código de error en el valor de retorno de la función o en uno de los argumentos de la llamada como un entero de 8 bits sin signo. Los códigos de error soportados por MicroC/OS-II se pueden encontrar en el fichero "*ucos_ii.h*" y contiene un total de 31 tipos diferentes de códigos de error. En la siguiente tabla 7.26, se pueden ver cuales han sido y en que porcentaje se han obtenido códigos de error tras la ejecución de los experimentos de inyección de fallos:

Tabla 7.26: Códigos de error devueltos por el SO

		Frecuencia	Porcentaje	Porcentaje válido	Porcentaje acumulado
Válidos	Error 1	33	36,3	36,3	36,3
	Error 10	1	1,1	1,1	37,4
	Error 20	17	18,7	18,7	56,0
	Error 40	31	34,1	34,1	90,1
	Error 42	9	9,9	9,9	100,0
	Total	91	100,0	100,0	

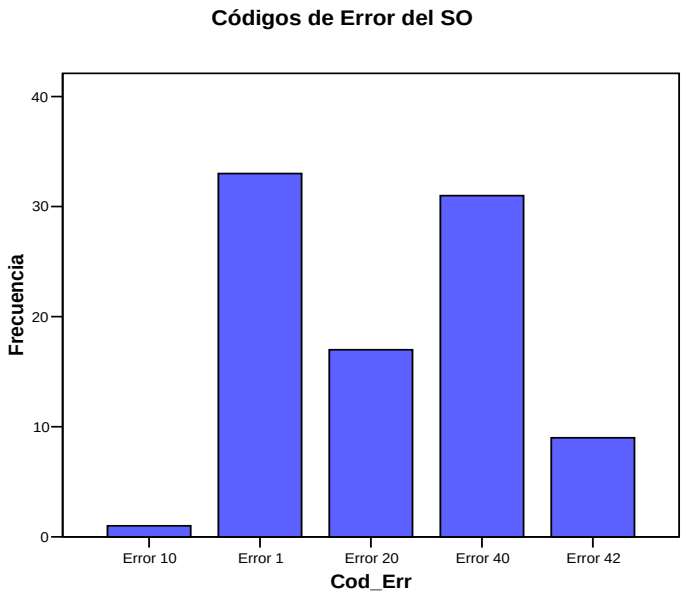


Figura 7.26: Frecuencia de obtención de los Códigos de Error

Clasificación y descripción de los códigos de error obtenidos:

- **Error 1**: Este código de error se codifica como "OS_ERR_EVENT_TYPE". Está relacionado con estructuras de tipo "OS_EVENT", que sirven para apuntar a colas de mensajes, semáforos, buzones y otros mecanismos de sincronización del SO. Se ha obtenido en un 41,9% de las veces. Llamadas relacionadas con este código de error son: OSMboxPend, OSMboxPost, OSMboxQuery, OSQFlush, OSQPend, OSQPost, OSQPostFront, OSQQuery, OSSemPend, OSSemPost y OSSemQuery. Este código de error se produce cuando el argumento "OS_EVENT *pevent" de estas llamadas al sistema no apunta correctamente a uno de los mecanismos de sincronización, dependiendo de la llamada.

- **Error 10**: Este código de error se codifica como "OS_TIMEOUT". Está relacionado con el manejo de buzones, colas y semáforos. Se ha obtenido en un 1,1% de las veces. Llamadas relacionadas con este código de error son: OSMboxPend, OSQPend y OSSemPend. Este código de error se produce cuando a un buzón o cola de mensajes no ha llegado ningún mensaje transcurrido un tiempo determinado. O en el caso de semáforos, cuando también transcurrido un tiempo determinado no se ha abierto el semáforo sobre el cual está definido el "timeout" o tiempo máximo de espera.

- **Error 20**: Este código de error se codifica como "OS_MBOX_FULL". Está relacionado directamente con la llamada al sistema OSMboxPost, que sirve para enviar un mensaje a un buzón. Se ha obtenido en un 18,7% de las veces. Este código de error se produce cuando se intenta enviar un mensaje a un buzón que ya contiene un mensaje.

- **Error 40**: Este código de error se codifica como "OS_PRIO_EXIST". Está relacionado con la creación de nuevas tareas en el sistema o con la asignación dinámica de prioridades a tareas ya existentes en el mismo. Se ha obtenido en un 12,9% de las veces y se produce cuando se intenta asignar a una tarea nueva o existente un valor de prioridad que ya existe para otra tarea que está funcionando en el sistema. Llamadas relacionadas con este código de error son: OSTaskChangePrio, OSTaskCreate y OSTaskCreateExt.

- **Error 42**: Este código de error se codifica como "OS_PRIO_INVALID". También está relacionado con la creación de nuevas tareas en el sistema o con la asignación dinámica de prioridades a tareas ya existentes en el mismo. Este error se produce cuando se intenta asignar un valor para la prioridad que se sale de rango o el valor nuevo a asignar es idéntico al que la tarea ya posee. Se ha obtenido en un 25,8% de las veces. Llamadas relacionadas con este código de error son: OSTaskChangePrio, OSTaskCreate, OSTaskCreateExt, OSTaskDel, OSTaskDelReq, OSTaskQuery, OSTaskResume, OSTaskSuspend y OSTimeDlyResume.

7.4.3 Excepciones

Una vez se han estudiado los códigos de error obtenidos por parte del SO, a continuación se va a analizar cuáles han sido las excepciones lanzadas por el microcontrolador MPC565, que se corresponde con aquellos casos en los que tras la inyección de un fallo, los mecanismos de detección de errores del microcontrolador son los que detectan un error en el sistema y en este caso lo llevan a finalizar la ejecución. Estos han sido codificados como salida "R3" del experimento y han supuesto un 35,1% del total de los experimentos.

En la figura 7.27 se puede observar qué excepciones se han obtenido y con qué frecuencia:

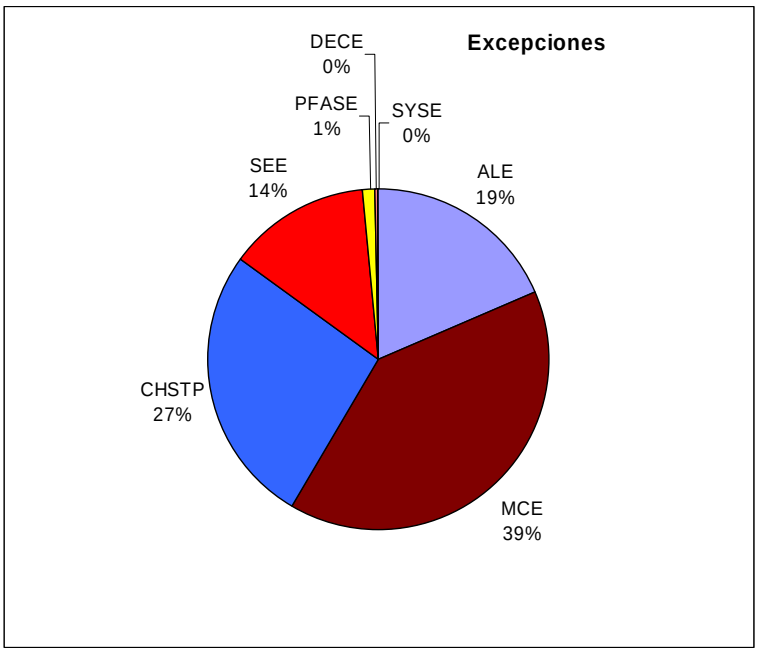


Figura 7.27: Frecuencia de obtención de las Excepciones

Como se puede apreciar en la figura 7.27, el mecanismo que más errores ha detectado es la excepción de tipo "*machine check*" (MCE). Estas excepciones no garantizan que se conserve el estado de los registros del procesador y, por lo tanto, se consideran no recuperables. En estos casos la única forma de recuperar el sistema sería mediante un reset. En segundo lugar y con un porcentaje también alto la excepción de "*checkstop*" (CHSTP), donde también se produce una suspensión en el procesamiento de instrucciones y el sistema debe ser reseteado para ser recuperado. En estos casos esto puede ser debido por al acceso a una dirección que no existe, o un error en los datos o una violación de protección de almacenamiento.

También se ha obtenido en un porcentaje del 19% los errores indicados con la excepción ALE, que son errores de alineamiento en la memoria, probablemente provocados por una mutación en algún operando accedido en modo indirecto. Con un 14% la excepción de detección de código de operación incorrecto, activando la excepción de "emulación software" (SEE), al intentar ejecutar instrucciones de acceso a la caché de datos o a registros de segmento, o instrucciones de acceso a registros de propósito especial.

Y en último lugar, las excepciones de tipo FPASE que indican errores detectados en la unidad de coma flotante, que se han producido en muy pocos casos. Al igual que las excepciones de DECE correspondiente al "*decrementer*", sólo en 3 ocasiones, y SYSE correspondiente a la ejecución de una instrucción de llamada al sistema errónea y que se ha producido en tan sólo 1 experimento.

7.4.4 Tiempos

En el siguiente apartado se va estudiar cómo las campañas de inyección de fallos han podido incidir en los resultados temporales que se esperan de un funcionamiento correcto del sistema. Como se ha adelantado en los primeros capítulos, las averías del sistema se pueden producir tanto por la obtención de valores incorrectos durante la ejecución del sistema, y por tanto se habla de averías en el dominio del valor, como por averías en el dominio del tiempo, por la obtención de resultados fuera de los límites temporales marcados como "*deadlines*" para la obtención de dichos valores correctos. Por tanto, a continuación se va a examinar los tiempos de

detección de errores del SO, los tiempos de las tareas de la aplicación en ejecución libre de errores, y la influencia de los fallos inyectados en los tiempos de ejecución de las mismas.

7.4.4.1 Tiempos de detección del SO

En este apartado se van a calcular los tiempos de detección de errores del SO, siguiendo la metodología anteriormente propuesta para la obtención de tiempos de latencia de detección. Para ello se obtiene la diferencia temporal entre el instante en el que el fallo se hace efectivo (se activa) en el sistema y el instante en el cual el mecanismo de detección de errores del SO en cuestión notifica un código de error indicando un estado erróneo del sistema. Así por tanto los datos obtenidos son (valores en microsegundos):

Tabla 7.27: Tiempos de detección de errores del SO

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Cod_Err 1	33	14,30	29,86	19,9848	,54787	3,14726	9,905
N válido (según lista)	33						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Cod_Err 10	1	21,94	21,94	21,9400	.	.	.
N válido (según lista)	1						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Cod_Err 20	17	20,24	134,46	47,3576	11,33881	46,75110	2185,666
N válido (según lista)	17						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Cod_Err 40	31	19,78	22,10	21,8516	,12380	,68927	,475
N válido (según lista)	31						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Cod_Err 42	10	12,78	12,80	12,7980	,00200	,00632	,000
N válido (según lista)	10						

En la tabla 7.27 se pueden observar los tiempos medios, valores máximos y mínimos obtenidos con el objetivo de calcular, cuál es el coste temporal que supone tener una notificación por parte del SO de que un error ha sido detectado durante la ejecución de la aplicación. De los estadísticos anteriores destacar que en este caso, y para la aplicación dada, los errores relacionados con la detección del envío de un mensaje a un buzón que ya estaba lleno, ha tenido un coste temporal mayor llegando a los 134,46us en los peores casos. Y con respecto al resto de códigos de error se puede apreciar que los tiempos de detección de errores están muy próximos en todos los casos entorno a los 20us, exceptuando el código de error 42 que no llega a los 13us.

A continuación y de un modo no tan específico se van a analizar los datos totales sobre los tiempos de detección de errores del SO en todos los casos. Así por tanto se tiene que (valores en microsegundos):

Estadísticos

N	Válidos	92
	Perdidos	43
Media		24,9120
Error típico de la media		2,35508
Mediana		21,7400
Moda		22,10
Desviación típica		22,58911
Varianza		510,268
Mínimo		12,78
Máximo		134,46

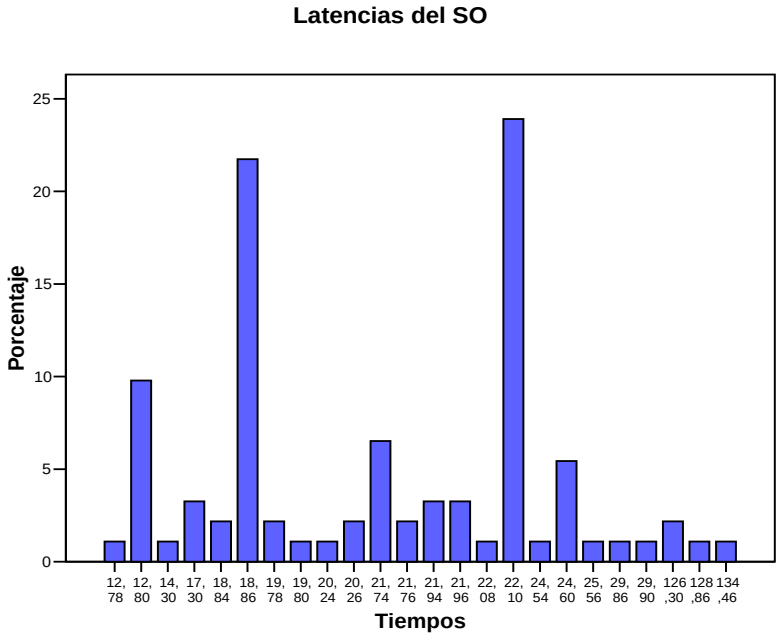


Figura 7.28: Frecuencia de las latencias de detección

Como se puede observar en los tiempos de detección de errores del SO, para la aplicación dada, hay dos valores que destacan con un mayor número de casos, como son los 18,86us y los 22,10us que suponen cerca del 46% de todos los casos obtenidos. También se observa que hasta los 18us hay también bastantes casos. Aunque la mayoría de los tiempos, y por tanto las latencias de detección del SO, se encuentran entre los dos valores anteriormente mencionados. En definitiva se puede afirmar que los tiempos de detección se encuentran en la mayoría de los casos en el rango que va de los [18,86us, 22,10us].

7.4.4.2 Tiempos de las tareas

A continuación en este apartado se va mostrar cual ha sido el efecto de los fallos inyectados con respecto a los tiempos de ejecución de las tareas. El objetivo es poder observar cómo la inyección de fallos de diseño del software, en un sistema constituido por componentes COTS, puede afectar a los propios componentes en el cumplimiento del trabajo asignado a cada una de las tareas del sistema dentro de los tiempos establecidos. Con esto se puede ver que a pesar de

que los valores obtenidos al final son correctos, los tiempos en los que estos valores son producidos pueden variar. Estas variaciones temporales de las que se habla podrán ser más o menos críticas dependiendo del criterio del diseñador de la aplicación a la hora de establecer los tiempos máximos asignados a las tareas, y que delimitarían las averías del sistema en el dominio del tiempo. En la siguiente tabla 7.28, se pueden ver las diferentes tareas que se ejecutan en el sistema, su función y el nombre que se les ha dado a la hora de identificar los tiempos obtenidos.

Tabla 7.28: Descripción de las tareas del sistema.

Tarea	Descripción
Control	Bucle de control de la gestión del tráfico para cada calle
SensorA	Bucle de control de la llegada/salida de vehículos en el semáforo A
SensorB	Bucle de control de la llegada/salida de vehículos en el semáforo B
SensorC	Bucle de control de la llegada/salida de vehículos en el semáforo C
SensorD	Bucle de control de la llegada/salida de vehículos en el semáforo D
TrafficIn	Bucle que simula la llegada de vehículos a cada una de las calles
TrafficOut	Bucle que simula la salida de vehículos de cada una de las calles

Los siguientes estadísticos representan en primer lugar los datos obtenidos correspondientes a los tiempos máximos de ejecución de las tareas para aquellos experimentos en los que no se inyectaban fallos (ejecuciones libres de fallos). En estos casos, dado que los resultados de los experimentos son deterministas se han tomado un conjunto muestras suficiente para establecer la referencia temporal de las tareas en ejecución libre de fallos. Seguidamente se muestran los resultados obtenidos para los tiempos máximos de ejecución de las tareas de la aplicación, pero una vez se han inyectado fallos en el sistema. En estos casos los tiempos que se obtienen se corresponden con aquellos experimentos que se han codificado como “R0”, donde se observaba que la inyección de un fallo finalmente no afectaba, desde el punto de vista del dominio del valor, a la entrega del servicio. Así por tanto, lo tiempos obtenidos tarea por tarea son los siguientes (valores en milisegundos):

Tabla 7.29: Tiempos para la tarea Control sin fallos y con fallos

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Control	115	2,876	2,876	2,87600	,000000	,000000	,000
N válido (según lista)	115						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
Control	590	2,872	3,011	2,98252	,001969	,047827	,002
N válido (según lista)	590						

Como se ha mencionado, en primer lugar se presentan datos correspondientes a las ejecuciones libres de errores y en segundo lugar los datos correspondientes a los experimentos donde se han inyectado fallos.

En primer lugar se va a analizar los resultados obtenidos para la tarea Control. Tal como muestran los estadísticos de la tabla 7.29, se puede observar que en ausencia de fallos, la tarea ha tenido un tiempo de ejecución bastante determinista de 2,876ms. Si se estudian los datos correspondientes a los experimentos bajo inyección de fallos, aquí se puede ver que ha habido una cierta variabilidad obteniendo un tiempo mínimo inferior en 4us y un tiempo máximo superior en 135us. La media también en este caso dista por encima unos 106us.

Tabla 7.30: Tiempos para la tarea SensorA sin fallos y con fallos

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
SensorA	115	2,892	3,020	3,00951	,003040	,032601	,001
N válido (según lista)	115						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
SensorA	590	2,892	3,033	3,00869	,001317	,031999	,001
N válido (según lista)	590						

Para el caso de la tarea SensorA se observa que en este caso los tiempos mínimos han sido idénticos en ausencia y presencia de fallos, por tanto se puede afirmar que esta tarea no ha podido provocar averías por entrega temprana del servicio. Con respecto a los valores máximos, en este caso en presencia de fallos, los tiempos se han incrementado en 13us, lo cual puede ser considerado como despreciable, aunque habrá que ver posteriormente si en la gráfica de distribución de tiempos se han dado muchos casos de éstos.

Tabla 7.31: Tiempos para la tarea SensorB sin fallos y con fallos

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
SensorB	115	2,879	3,012	2,98485	,002926	,031383	,001
N válido (según lista)	115						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
SensorB	590	2,879	3,012	2,97133	,001966	,047751	,002
N válido (según lista)	590						

Para el caso de la tarea SensorB, se puede apreciar que tanto los tiempos mínimos como máximos son equivalentes en ambos casos, y aunque la media en presencia de fallos se incrementa en algunos microsegundos, en este caso no tiene relevancia, ya que no hay valores en sus extremos que se salgan del rango de tiempos en el cual la tarea se ha ejecutado en ausencia de fallos. Por tanto se puede asegurar que la tarea SensorB no se ha visto en absoluto influenciada en sus tiempos de ejecución por los fallos inyectados.

Tabla 7.32: Tiempos para la tarea SensorC sin fallos y con fallos

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
SensorC	115	2,881	3,010	2,99219	,003661	,039261	,002
N válido (según lista)	115						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
SensorC	578	2,878	3,027	2,97306	,002274	,054662	,003
N válido (según lista)	578						

En cuanto a la tarea SensorC, de los estadísticos de la tabla 7.32 se puede observar que en este caso los valores mínimos de los tiempos de ejecución en presencia de fallos han llegado en algún caso a adelantarse en 3us, lo cual también es casi despreciable. Y en cuanto a los tiempos máximos, algún caso se ha retrasado en 17us, lo cual también es poco significativo, aunque esta apreciación a priori dependería claramente de los requisitos establecidos por el diseñador para los tiempos de ejecución de las tareas.

Tabla 7.33: Tiempos para la tarea SensorD sin fallos y con fallos

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
SensorD	115	2,882	3,014	2,94304	,005656	,060659	,004
N válido (según lista)	115						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
SensorD	590	2,716	3,029	2,95254	,002634	,063985	,004
N válido (según lista)	590						

Para el caso de la tarea SensorD se observa que los tiempos mínimos en caso de fallos se adelantan en 166us y para los tiempos máximos, también en caso de fallos, se retrasan en 15us. Y la media dista en tan sólo 9us.

Tabla 7.34: Tiempos para la tarea TrafficIn sin fallos y con fallos

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
TrafficIn	115	3,096	3,214	3,20668	,002116	,022686	,001
N válido (según lista)	115						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
TrafficIn	590	3,079	3,362	3,19843	,001629	,039558	,002
N válido (según lista)	590						

Para la tarea TrafficIn, tarea encargada de simular el tráfico de llegada. Los datos obtenidos muestran que la media en caso de fallos es inferior en 8us, lo cual no es mucho, pero si que ha habido casos en los cuales la tarea ha finalizado su ejecución 17us antes. Aunque más significativo es que ha habido casos en los cuales la tarea ha finalizado 148us después retardando en este caso la llegada de vehículos que la tarea simula. Ello ha podido producir que la tasa esperada de llegada de vehículos no haya sido la misma, y por tanto el resto de tareas hayan tenido que esperar más de la cuenta para terminar sus respectivos ciclos de ejecución.

Tabla 7.35: Tiempos para la tarea TrafficOut sin fallos y con fallos

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
TrafficOut	115	2,819	2,951	2,94124	,003131	,033579	,001
N válido (según lista)	115						

Estadísticos descriptivos

	N	Mínimo	Máximo	Media		Desv. típ.	Varianza
	Estadístico	Estadístico	Estadístico	Estadístico	Error típico	Estadístico	Estadístico
TrafficOut	588	2,749	2,954	2,93251	,001832	,044413	,002
N válido (según lista)	588						

En último lugar para la tarea TrafficOut, que se encarga de cambiar el estado de los semáforos y por tanto, desde el punto de vista del servicio que ofrece la aplicación es la tarea más crítica. Para ésta se observa que los valores mínimos distan en 70us, lo que significa que en caso de fallos en alguno o en algunos experimentos, aunque esto se verá mejor con las gráficas de distribución de tiempos, la tarea ha cambiado el estado de un semáforo antes de tiempo. Y con respecto a los tiempos máximos, que en este caso supondría que los vehículos tienen que esperar más tiempo a que el semáforo cambie, la diferencia en este caso es de tan sólo 3us.

7.4.4.3 Distribución de los tiempos máximos

En al apartado anterior se han estudiado los valores obtenidos de los estadísticos producidos a partir de la evaluación de los tiempos máximos de ejecución de las tareas. A continuación se va analizar, cuál es la distribución de esos tiempos máximos para cada una de las tareas en los experimentos que se han llevado a cabo. El objetivo es poder observar cual de éstas se han visto mayormente afectada en sus tiempos de ejecución a causa de los fallos introducidos y cuales de éstas serían más propensas a introducir retardos o adelantos en el tiempo total de ejecución del sistema.

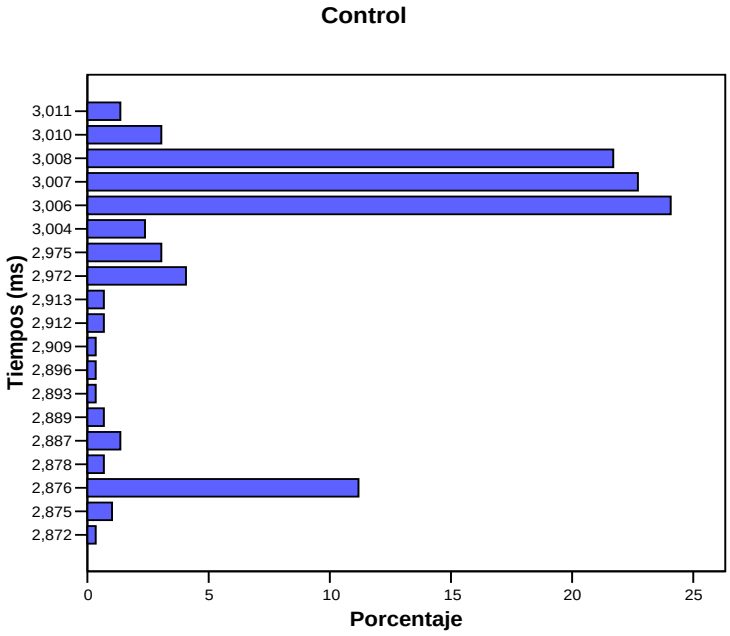


Figura 7.29: Distribución de los tiempos máximos para la tarea Control

En primer lugar con respecto a la distribución de tiempos para la tarea Control. Como se puede observar en la gráfica de arriba correspondiente a la figura 7.29, se puede apreciar claramente que por encima de los 2,876us, que es lo que debería durar la ejecución de la tarea en condiciones normales, tan sólo se ha alcanzado un 12% de las veces, observando que por encima de este tiempo están cerca del 85% del resto de los experimentos. Esto lleva a la conclusión de que parece ser que esta tarea se ha visto bastante afectada en sus tiempos de ejecución por el efecto de los fallos introducidos en el sistema, provocando bastantes averías por retardar considerablemente la entrega del servicio.

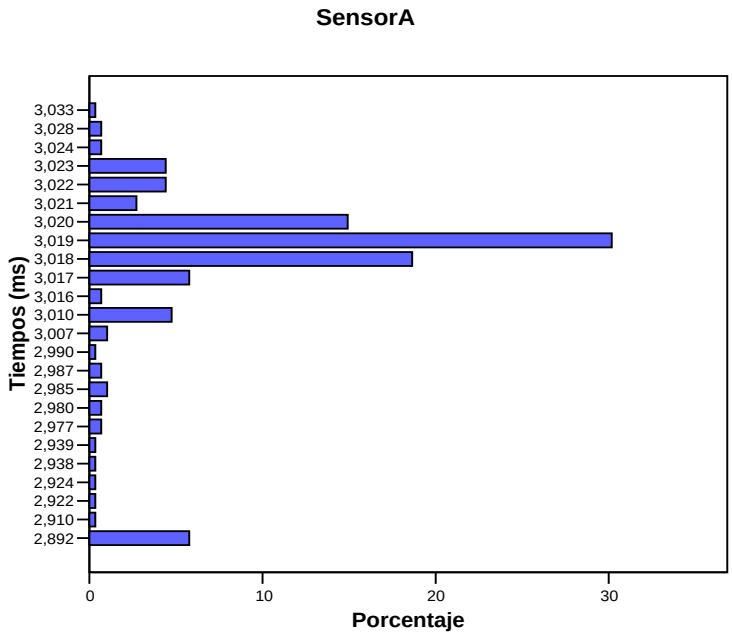


Figura 7.30: Distribución de los tiempos máximos para la tarea SensorA

Con respecto a la tarea SensorA se puede observar que en este caso por debajo de 2,892ms, que era el límite inferior para dicha tarea, no existen experimentos, por tanto ausencia de averías por entrega temprana del servicio. Por otro lado con respecto al límite superior, en este caso se puede ver que cerca del 14% de las veces se han obtenido tiempos por encima del valor máximo estimado para una ejecución normal de la tarea.

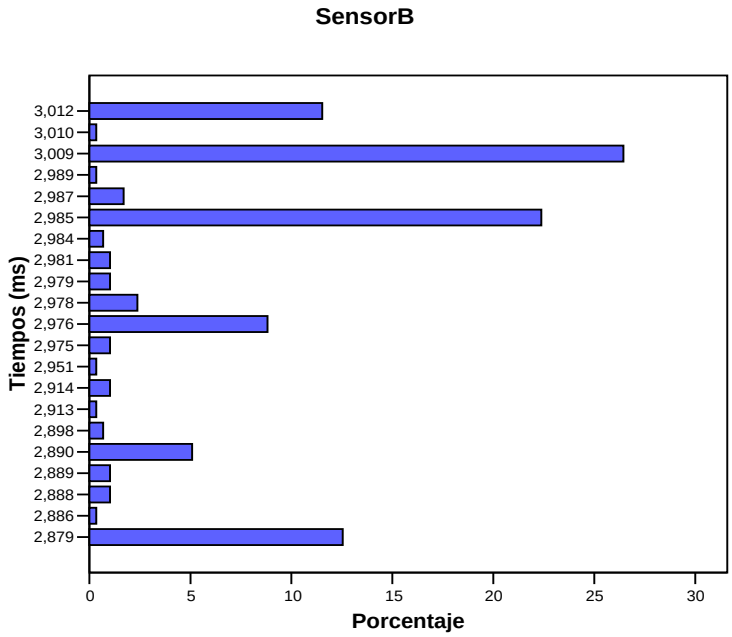


Figura 7.31: Distribución de los tiempos máximos para la tarea SensorB

En el caso de la tarea SensorB claramente y como ya se ha comentado anteriormente, se puede observar que todos los tiempos de ejecución recogidos de los experimentos han estado en el intervalo [2,879ms, 3,012ms], no obteniendo por tanto ninguna avería con respecto al tiempo de entrega del servicio. Se puede volver a afirmar que en este caso la tarea no se ha visto afectada en absoluto en sus tiempos de ejecución por causa de los fallos inyectados.

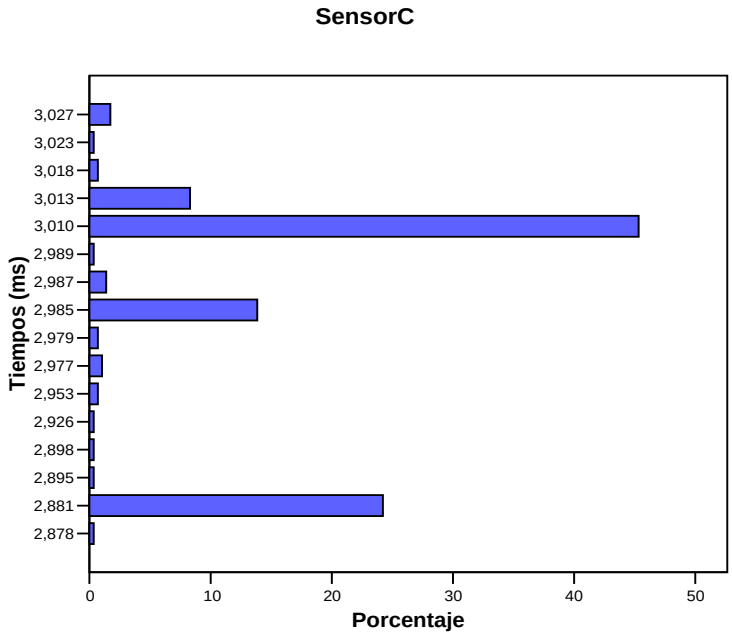


Figura 7.32: Distribución de los tiempos máximos para la tarea SensorC

Con respecto a la tarea SensorC se puede apreciar que por debajo de los 2,881ms se tienen muy pocos experimentos, éstos se corresponderían con averías por entrega temprana del servicio. Y por encima de los 3,010ms se ha obtenido cerca de un 10% de casos, donde el tiempo de ejecución ha sido mayor de lo esperado para una ejecución libre de errores. Estos casos se corresponderían con averías por entrega tardía del servicio.

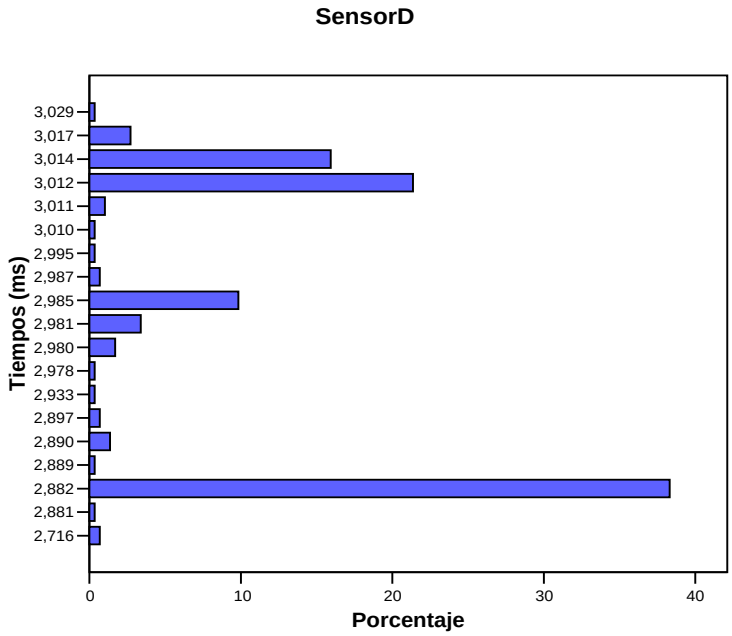


Figura 7.33: Distribución de los tiempos máximos para la tarea SensorD

Para la tarea SensorD, según se puede ver en la gráfica de la figura 7.33, se tiene que en este caso se puede hablar de que tan sólo alrededor de un 5% de los tiempos obtenidos se salen del rango de [2,882ms, 3,014ms]. Es decir se puede ver que en definitiva la tarea no se ha visto prácticamente afectada por el efecto de los fallos introducidos. En este caso se tiene que en cerca del 95% de los experimentos la tarea se ha ejecutado dentro del intervalo de tiempos considerado como correcto en condiciones normales de ejecución.

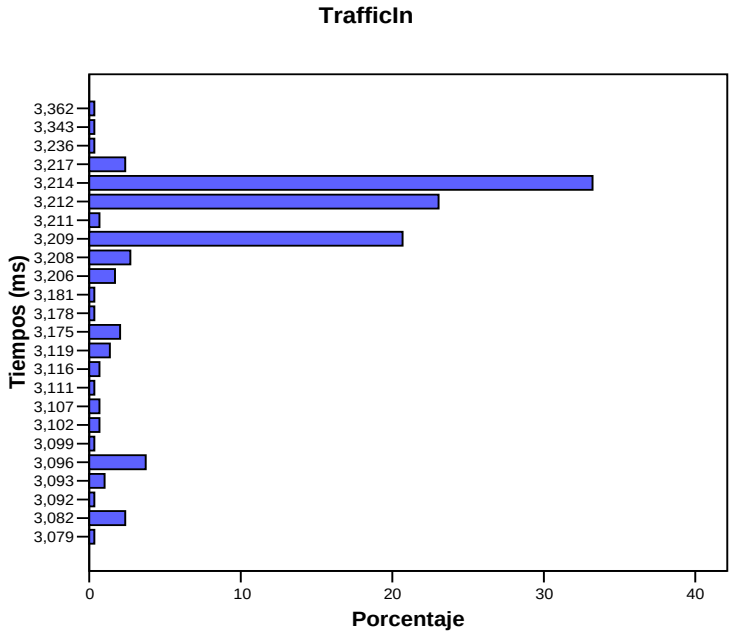


Figura 7.34: Distribución de los tiempos máximos para la tarea TrafficIn

Con respecto a la tarea TrafficIn según se puede observar de la figura 7.34. Se tiene que por debajo de 3,096ms hay alrededor de un 5% de casos que se pueden considerar como averías por entrega tardía del servicio. Y por encima de los 3,214ms también se puede apreciar que hay alrededor de otro 5% de experimentos, en los que se ha alargado el tiempo de ejecución de dicha tarea. Con lo que se puede concluir que cerca de un 10% de las veces se ha obtenido tiempos de ejecución que se salen del intervalo de tiempos adecuado para esta tarea.

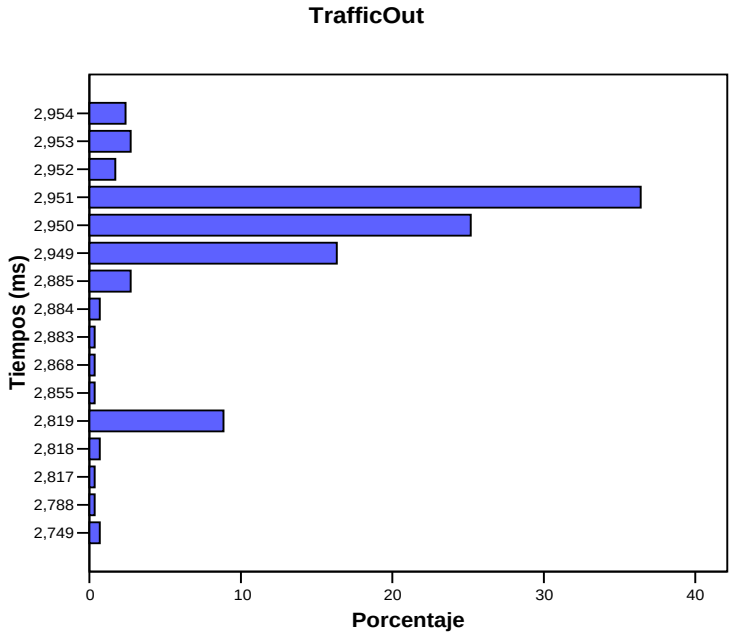


Figura 7.35: Distribución de los tiempos máximos para la tarea TrafficOut

Finalmente para la tarea TrafficOut, que como se ha indicado anteriormente es la encargada de cambiar el estado de los semáforos. Se tiene que por debajo de los 2,819ms se puede encontrar alrededor de un 4% de casos en los que la tarea se ha adelantado, casos de averías por entrega temprana del servicio. Y por encima de los 2,951ms, alrededor de un 10% de casos que

supondrían averías por retraso en la entrega del servicio. Es decir, averías efectivas habrían alrededor de un 14% del total de los casos en los cuales, a pesar de que la aplicación en cuanto a valores producidos en sus salidas ha sido correcta, en cuanto al tiempo no ha ocurrido lo mismo, entregando los valores antes o después de lo esperado.

7.5 DISCUSIÓN DE RESULTADOS SOBRE AMBOS SISTEMAS OPERATIVOS

Una vez se han presentado los resultados que se han obtenido de las diferentes campañas de inyección de fallos, que se han llevado a cabo para cada una de las tres cargas que se han utilizado. A continuación a modo de discusión se van a estudiar los resultados que se han obtenido. El objetivo es poder extraer conclusiones acerca de qué sistema operativo nos daría una mayor confiabilidad para una aplicación dada, que en este caso de análisis se corresponde con la que simula el control de semáforos en un cruce; aplicación que ha sido probada con ambos sistemas operativos.

En primer lugar y observando los datos relativos a la finalización de la ejecución de la aplicación en los diferentes experimentos, se observa que para OSEK en un 40,9% de las veces los fallos introducidos han sido tolerados frente al 30,4% que se ha obtenido para MicroC/OS-II. También significativo es que para OSEK tan sólo se han obtenido un 13,2% de experimentos, donde la ejecución finalmente fue abortada por el lanzamiento de una excepción por parte del microcontrolador, frente al 35,1% de excepciones que se obtuvo para MicroC/OS-II. En este último caso, observamos que el microcontrolador fue más rápido que el sistema operativo en detectar los errores introducidos.

En cuanto a averías del sistema, si se habla de valores de salida producidos por la aplicación (dominio del valor), también se observa que mientras que para OSEK se ha obtenido cerca de un 60% de averías, de éstas el 47,8% han quedado cubiertas por los mecanismos de detección del propio sistema operativo. En MicroC/OS-II si bien se han obtenido un 34,5% de averías, en este caso tan sólo en el 6% de las mismas, el sistema operativo ha sido quien ha detectado el error frente a un 94% de las veces que no lo ha hecho. En este caso, y como se ha comentado anteriormente, se puede afirmar que OSEK ha detectado muchos más errores que MicroC/OS-II. Es por ello que además en el caso de OSEK, si el tratamiento de los códigos de error se hubiera implementado, posiblemente el número de averías se hubiera reducido considerablemente. En este caso simplemente hay una notificación de errores por parte del sistema operativo echando en falta el tratamiento de los mismos por parte de la aplicación. Así, con los datos en la mano finalmente se observa que la cobertura de errores de OSEK ha sido del 65,63% frente a la de MicroC/OS-II que ha sido del 32,5%.

En cuanto los tiempos de detección de errores del sistema operativo, para OSEK y dados los errores que se han obtenido, se tiene que dichos tiempos de detección han oscilado en el rango de [27,82us, 30,1us], mientras que para MicroC/OS-II la mayoría de los tiempos que se han obtenido se hallan en el rango de [18,86us, 22,10us]. En este caso se observa, que para los códigos de error que se han obtenido para cada uno de los sistemas operativos, los tiempos de detección de errores, aunque próximos para ambos RTOS, en el caso de MicroC/OS-II han sido inferiores a los de OSEK.

Finalmente, en cuanto los tiempos de ejecución de las tareas, se tienen datos suficientes para realizar una comparativa tarea a tarea para cada uno de los dos sistemas operativos, pero en este caso el análisis se centrará en la tarea denominada como "*TrafficOut*" por ser la más crítica. Tal criticidad viene dada porque dicha tarea, como se ha comentado ya en varias ocasiones, es la tarea encargada de producir los valores de salida de la aplicación, que en este caso se

corresponde con los valores que adoptan los semáforos en un momento determinado, dando o no paso a los diferentes vehículos que esperan en la intersección. En este caso se observa que para OSEK se tiene que la tarea *TrafficOut* ha producido alrededor de un 7% de tiempos en los cuales los valores salidas se han ofrecido antes o después de lo esperado. Mientras que para MicroC/OS-II esto ha ocurrido en un 14% de las veces. Teniendo en cuenta además que el porcentaje de experimentos donde la ejecución ha funcionado correctamente, produciéndose valores de salida correctos, en OSEK ha sido superior a MicroC/OS-II, finalmente se puede concluir que desde el punto de vista temporal también con OSEK se han producido menos averías que con MicroC/OS-II.

7.6 RESUMEN Y CONCLUSIONES

En este capítulo se han presentado los resultados de los experimentos llevados a cabo en inyección de fallos utilizando la herramienta que se ha desarrollado, y que se ha explicado el capítulo anterior, con dos objetivos principalmente: el primero de ellos consistía en validar experimentalmente la metodología que se proponía en el capítulo 5, para inyectar fallos de diseño del software en sistemas empotrados basados en componentes COTS, utilizando para ello las características que ofrece la interfaz Nexus™, para llevar a cabo una inyección de fallos y posterior observación de resultados de forma no intrusiva. Por otro lado también se pretendía ver si con el planteamiento del que se partía, era posible obtener datos objetivos para la evaluación de diferentes componentes COTS que pudieran ser integrados en un sistema para una aplicación determinada.

Así por tanto, se han presentado resultados para tres cargas diferentes. En primer lugar se ha utilizado una carga que consistía en el sistema operativo MicroC/OS-II más una aplicación que simula la unidad de control electrónica (ECU) de un motor diésel. En segundo lugar, se han utilizado otras dos cargas que utilizaban una misma aplicación de control de semáforos sobre un cruce, pero funcionando sobre dos sistemas operativos de tiempo real diferentes como son OSEK y MicroC/OS-II.

Con éstas se han obtenido datos relativos a coberturas de detección de errores, averías del sistema, códigos de error detectados, significado de los mismos y frecuencias. Por otro lado, también se han obtenido las excepciones que ha lanzado el microcontrolador y sus posibles causas, para finalmente evaluar información de tipo temporal. Dicha información de tipo temporal, ha aportado datos relativos a las averías, desde el punto de vista del dominio del tiempo, viendo los tiempos máximos de ejecución que alcanzaban las tareas cuando eran inyectados fallos en el sistema. De esta manera se han podido calcular los tiempos de detección de errores de los dos sistemas operativos y las posibles averías, por adelanto o retraso, en la entrega del servicio que al final la aplicación ofrece. Para finalizar se ha ofrecido una pequeña discusión a modo de comparativa sobre los datos obtenidos para la aplicación de semáforos funcionando sobre dos sistemas operativos diferentes.

Capítulo 8

CONCLUSIONES Y TRABAJO FUTURO

8.1 INTRODUCCIÓN

Finalmente, en este último capítulo se van a presentar las conclusiones que se pueden extraer de la presente tesis, resumiendo todo el trabajo realizado y planteando el conocimiento que de ésta se desprende. En primer lugar se plantea brevemente el porqué de la necesidad de encontrar una metodología y técnica de inyección de fallos que permita evaluar componentes COTS de forma no intrusiva en sistemas empotrados de tiempo real. Posteriormente, se presentan las conclusiones que se han obtenido a partir de los experimentos de inyección de fallos a través de la utilización de los mecanismos de depuración internos de los microprocesadores actuales. En este sentido se muestra la utilidad de la interfaz Nexus™ para llevar a cabo dicha inyección de fallos, con las campañas que se ha llevado a cabo y los resultados que de éstas se han obtenido.

8.2 CONCLUSIONES

Es un hecho más que constatable que la utilización de componentes software de muy diverso origen en el desarrollo de las aplicaciones y sistemas actuales viene siendo una práctica habitual. La reducción en el tiempo de desarrollo y costes asociados son dos de las grandes ventajas que esta práctica aporta a las organizaciones. Pero la integración de éstos en el desarrollo de los sistemas es todavía una cuestión que preocupa a la comunidad científica, sobretodo cuando se habla de sistemas críticos, donde el funcionamiento incorrecto de los sistemas informáticos pueden provocar serios daños personales o graves consecuencias económicas.

Si se habla de sistemas empotrados de tiempo real que deben ser tolerantes a fallos, la confiabilidad de los componentes que integran dichos sistemas debe ser medida. Es por ello la necesidad de encontrar metodologías y técnicas que permitan la evaluación de tales componentes. Pero tal evaluación debe ser realizada con los sistemas funcionando en un entorno real, para así poder obtener datos que sean representativos del funcionamiento de los sistemas y poder certificarlos, como así proponen los estándares y normas internacionales.

Es cierto que los componentes software deben ser verificados y validados antes ser utilizados, pero en muchas ocasiones dichos tests no confrontan al software a todas las posibilidades de operación que se puedan dar y es por ello que siempre quedan fallos de tipo residual en los mismos. Es bien conocido que la mayor parte de los fallos activados son debidos al software. En este sentido las técnicas de inyección de fallos se constituyen como un medio para evaluar el comportamiento del software ante la aparición de fallos, y así poder descubrir fallos en el diseño del mismo que todavía no han sido detectados en fases anteriores.

En la presente tesis se han evaluado diferentes componentes COTS, en particular se han testeado dos sistemas operativos diferentes sobre los que funcionaba una misma aplicación. En un caso se ha evaluado una implementación de la especificación OSEK/VDX, estándar ampliamente utilizado y apoyado por el sector de la automoción, además de estandarizado por ISO en la norma ISO17356. Y en el otro caso se ha utilizado el RTOS MicroC/OS-II, que ha sido certificado para sistemas críticos por las normas RTCA DO-178B, EUROCAE ED-12B, FDA510(k) e IEC61058. De dicha experimentación se ha obtenido que la inclusión de uno de ellos hizo que el sistema fuera capaz de tolerar el 41% de los fallos inyectados frente al 30,4% del otro. Que en uno de ellos se obtuvieran un 13% de excepciones por parte del microcontrolador frente a un 35% con respecto al otro. O que por ejemplo, del total de las averías un 47,8% de las mismas en un sistema quedarán cubiertas frente al 6% en el otro caso. Finalmente, se ha obtenido una cobertura de detección de errores de un 65,6% para un sistema frente a un 32,5% para el otro. En cuanto a aspectos temporales, que pudieran ser de interés para los diseñadores/usuarios de sistemas empotrados de tiempo real, también se ha podido observar los tiempos de detección de errores para ambos sistemas operativos, y el efecto de los fallos inyectados en los tiempos de ejecución de las tareas, donde se observa que del total de las averías en un sistema operativo un 7% de las mismas se debieron a entregas tempranas o tardías del servicio frente a un 14% en el otro caso.

De todo esto se observa el interés de realizar estudios de robustez de los componentes software y demuestra la necesidad de seguir investigando en nuevas metodologías, herramientas y técnicas para profundizar en el análisis de dichos componentes software que pudieran ser utilizados para el desarrollo de sistemas. Así en la presente tesis se ha planteado la necesidad de encontrar una metodología de inyección de fallos que permita verificar el funcionamiento, principalmente ante fallos de diseño del software, de un sistema construido a partir de la unión de diferentes componentes COTS que interactúan entre sí. Y consecuentemente, ha surgido la necesidad de desarrollar una herramienta de inyección de fallos que implemente dicha metodología para evaluar a los sistemas frente a la aparición de fallos en su funcionamiento real.

8.2.1 Técnicas Inyección de fallos

Para realizar dicho trabajo, en primera instancia se han estudiado las técnicas y herramientas de inyección de fallos que hay desarrolladas hasta la fecha. Estas técnicas se suelen agrupar en tres tipos, las basadas en inyección sobre modelos del sistema como la simulación (SBFI del inglés "*Simulation-Based Fault Injection"*) o la emulación, las que utilizan un hardware específico (HWIFI del inglés "*HardWare Implemented Fault Injection*") y las basadas en software (SWIFI del inglés "*SoftWare Implemented Fault Injection"*).

Las técnicas basadas en simulación utilizan un modelo del sistema e inyectan fallos mediante su simulación. Su principal ventaja es su aplicabilidad en la fase de diseño, gracias a lo que se pueden reducir o evitar los costes derivados de un rediseño del prototipo. El principal inconveniente es el elevado coste temporal de las campañas de inyección.

Las técnicas basadas en hardware inyectan fallos físicos en el hardware del sistema. Como principal ventaja cabe mencionar la alta representatividad de los fallos inyectados respecto a los reales. Como principal inconveniente tienen un coste elevado y unos modelos de fallos limitados.

La última solución consiste en emular tanto los fallos hardware y sus consecuencias como los fallos de diseño de los programas mediante software. Las ventajas de estas técnicas son su reducido coste de implementación y su facilidad de automatización. Como inconvenientes cabe mencionar la baja portabilidad de las herramientas de inyección, el modelo de fallos limitado, al no tener acceso a todos los registros del sistema y la alta intrusión temporal sobre el sistema que implica su utilización.

Es este último inconveniente, el de la intrusión temporal, el que limita la utilización de las técnicas SWIFI en los sistemas de tiempo real. Las especificaciones temporales del sistema (por ejemplo los plazos de las tareas) se pueden incumplir por culpa de la intrusión adicional que crea la introducción de la instrumentación de una herramienta de inyección de fallos software. Este es un problema importante que se debe tener en cuenta siempre que se utilicen herramientas SWIFI en un sistema de tiempo real, de manera que se asegure que no se introduce ninguna distorsión temporal en la fase de verificación y validación que perjudique la exactitud de los resultados que se obtienen.

Así pues, se han estudiado los trabajos realizados utilizando herramientas SWIFI en sistemas de tiempo real y aparecen dos posibles soluciones en busca de una inyección de fallos no intrusiva. La primera consiste en parar el reloj del sistema cada vez que se ejecuta código de la herramienta de inyección y reiniciarlo al volver a ejecutar código del sistema. De esta manera la aplicación se ejecuta en un entorno virtual en el que no existe interferencia de la herramienta, aunque esta solución no se puede aplicar en un sistema real, ya que aparecería un desfase entre el reloj del sistema y el tiempo real de su entorno. La segunda solución implica un estudio detallado de la aplicación y del sistema operativo que la ejecuta. Se diseña un sistema limitado de monitorización de la aplicación en el que se reduce la intrusión hasta el mismo orden de magnitud que la latencia de la interrupción del núcleo de tiempo real y dentro de la dispersión en los tiempos de ejecución de la carga que se esté ejecutando.

Por otro lado, también se ha realizado una búsqueda de trabajos de inyección de fallos orientados específicamente a la inyección sobre sistemas basados en microcontroladores o SoCs (del inglés "*Systems on a Chip*"). Esta búsqueda se ha llevado a cabo debido a que estos sistemas presentan una dificultad adicional por las limitaciones en la observabilidad y controlabilidad que impone el tener tanto procesador como memorias y periféricos dentro del mismo encapsulado.

Así pues, de este trabajo de búsqueda se planteó como objetivo la propuesta de una técnica que fuera capaz de inyectar fallos de diseño del software en sistemas basados en microcontroladores sin introducir ningún tipo de distorsión en el sistema bajo prueba.

8.2.2 Inyección de fallos basada en Nexus™

Dentro del estudio de los microcontroladores actuales se han estudiado los mecanismos disponibles en ellos que pueden ayudar a conseguir el objetivo de una inyección de fallos sin ningún tipo de intrusión. En ese sentido se ha estudiado Nexus 5001™, que es un estándar aceptado por el IEEE para la unificación de las herramientas de depuración de sistemas basados en microcontroladores. En Nexus™ se propone la estandarización de los mecanismos de depuración que los fabricantes de microcontroladores han ido incluyendo en los chips en los

últimos años. De esta manera con una sola herramienta de depuración se puede trabajar con sistemas implementados como SoC de cualquier fabricante.

Más allá de su interés como estándar de depuración, se ha propuesto utilizar Nexus™ para realizar la inyección de fallos. Así pues, las ventajas que puede aportar Nexus™ a dicho proceso son las siguientes:

- Portabilidad de la herramienta de inyección; al usar un estándar como Nexus™ se podrán inyectar fallos sobre cualquier sistema que sea compatible con él.
- Observabilidad; como Nexus™ se ha definido precisamente para depuración de sistemas basados en microcontroladores aporta mecanismos para mejorar la observación de los mismos.
- Nula intrusión temporal sobre el sistema; gracias a la característica de Nexus™ de acceso al sistema en tiempo de ejecución. Esta característica permite leer y escribir, y por tanto modificar, cualquier posición en el mapa de memoria del sistema sin alterar la temporización en la ejecución de las instrucciones.

Así con las ventajas que ofrece Nexus™, se ha propuesto una metodología para realizar la inyección consistente en:

- Un modelo de fallo, que puede ser la inversión o *bit-flip* de uno o varios bits.
- Una sincronización de la inyección, que puede ser temporal, espacial o una combinación de un disparo espacial seguido por uno temporal.
- Una localización del fallo, que se corresponde con los parámetros de las funciones que implementan la API de los componentes.

Para ello, la metodología que se ha presentado se propone en una sucesión de pasos que establecen el proceso por el cual, a partir de una aplicación dada se obtienen las direcciones correspondientes a los parámetros de las llamadas de los componentes donde realizar la inyección de fallos y observar la robustez de dichos componentes frente a fallos externos provenientes de otros. Además siempre bajo una perspectiva de los mismos de caja negra, donde se supone que no se tiene acceso al código fuente de los componentes.

Pero en la metodología propuesta, no sólo se plantean cuestiones referentes al proceso de inyección de fallos de diseño del software, sino también se proponen una serie de pasos para establecer los mecanismos necesarios de observación del efecto que finalmente producen los fallos introducidos en los componentes, y así poder observar fallos activados, errores y finalmente averías tanto en el dominio del tiempo como del valor. Para ello se ha estudiado qué medidas se pueden obtener y cómo obtenerlas a partir de la información que dan las trazas de Nexus™.

Así pues, para cada experimento se puede obtener si se ha activado el fallo inyectado, si se ha detectado un error, si el error se ha propagado a una avería, el instante de activación del error, el instante de detección del error y los tiempos, en el peor de los casos, de la ejecución de las tareas del sistema para observar averías que se producen en el no cumplimiento de los tiempos de entrega del servicio.

En resumen, se ha propuesto y validado un metodología para abordar el problema de la integración de componentes especificando punto a punto los pasos a seguir para realizar la inyección de fallos de diseño del software, y así obtener una evaluación de la robustez y comportamiento temporal de componentes COTS integrados en sistemas empotrados para aplicaciones de tiempo real.

8.2.3 Herramienta de inyección realizada

Una vez se estudiaron las capacidades disponibles en la interfaz Nexus™, y posteriormente se propuso una metodología de inyección de fallos de diseño del software y monitorización de los mismos, que no provocará intrusión alguna en el sistema bajo evaluación, se ha desarrollado una herramienta que implementa dicha metodología y permite la observación de los resultados provenientes de las campañas de inyección de fallos.

La herramienta está formada por tres módulos: el generador de experimentos, el inyector de fallos y el módulo de análisis. El generador de experimentos realiza un primer análisis para establecer los parámetros que definirán el proceso de inyección, posteriormente el inyector de fallos lleva a cabo los diferentes experimentos de inyección de fallos, y finalmente el módulo de análisis se encarga de devolver los resultados de la experimentación, en cuanto a coberturas de detección de errores, latencias de detección, y de activación de errores y averías, y tiempos de ejecución de las tareas bajo el efecto de los fallos inyectados.

La herramienta desarrollada ha permitido verificar y validar la metodología propuesta para inyectar fallos de diseño del software, aunque con ésta también es posible inyectar fallos físicos en cualquier zona de memoria, siendo este proceso más simple. La herramienta permite inyectar fallos tanto en tiempo de ejecución (*runtime*), con diferentes posibilidades de sincronización en el disparo, como antes de lanzar a ejecución el sistema (*pre-runtime*), permitiendo además inyectar fallos de diferentes tipos. Finalmente, la herramienta hace uso de los mecanismos que Nexus™ dispone para observar en tiempo real el funcionamiento del sistema sin introducir intrusión alguna.

8.2.4 Resultados de los experimentos

Finalmente, tras el desarrollo de la herramienta se han llevado a cabo una serie de experimentos con el objetivo de verificar y validar experimentalmente la metodología propuesta. Para ello se han utilizado tres cargas diferentes que han validado que la metodología mostrada es independiente del componente COTS bajo estudio y del compilador que en su caso se haya utilizado.

En primer lugar se ha utilizado una carga que consistía en el sistema operativo MicroC/OS-II más una aplicación que simula la unidad de control electrónica (ECU) de un motor diésel. En segundo lugar, se han utilizado otras dos cargas que utilizaban una misma aplicación de control de semáforos sobre un cruce, pero funcionando sobre dos sistemas operativos de tiempo real diferentes como son OSEK y MicroC/OS-II.

Con éstas se han obtenido datos relativos a coberturas de detección de errores, averías del sistema, códigos de error detectados, significado de los mismos y frecuencias. Por otro lado, también se han obtenido las excepciones que ha lanzado el microcontrolador y sus posibles causas, para finalmente evaluar información de tipo temporal. Dicha información de tipo temporal, ha aportado datos relativos a las averías, desde el punto de vista del dominio del tiempo, viendo los tiempos máximos de ejecución que alcanzaban las tareas cuando eran inyectados fallos en el sistema. De esta manera se han podido calcular los tiempos de detección de errores de los dos sistemas operativos y las posibles averías, por adelanto o retraso, en la entrega del servicio que al final la aplicación producía. Para finalizar, se ha ofrecido una pequeña discusión a modo de comparativa sobre los datos obtenidos para la aplicación de semáforos funcionando sobre ambos sistemas operativos.

Todos estos resultados mostrados a partir de la experimentación validan la idea original que motivó la realización de la presente tesis en encontrar una nueva metodología de inyección de fallos no intrusiva para evaluar la robustez de componentes COTS frente a fallos de diseño, proveyendo, a nuestro entender, de una información muy interesante para los diseñadores de aplicaciones de sistemas empotrados de tiempo real. Debido a que poder obtener datos objetivos relativos al efecto de los fallos de diseño en componentes COTS, que puedan inducir errores en otros y que puedan finalmente llevar al sistema a producir averías, es de vital importancia.

Además, la característica de no intrusión de la metodología propuesta, aporta una mayor confianza en la representatividad de los resultados que se obtienen, sobretodo cuando se habla del aspecto temporal en sistemas de tiempo real con fuertes restricciones. Por último, importante también es desde el punto de vista del diseñador de sistemas, el poder obtener datos relativos al coste temporal que supone detectar errores en el mismo y así poder acotar los tiempos de recuperación del sistema. Y en último lugar, observar el efecto de los errores en los componentes sobre los plazos de ejecución de las tareas, o *deadlines* establecidos, permite obtener información relativa a la incidencia que los fallos de diseño de los componentes pueden acarrear a la temporización final de todo el sistema.

8.3 TRABAJO FUTURO

Las tecnologías basadas en componentes están evolucionando muy rápidamente, y actualmente ofrecen la posibilidad de desarrollar soluciones empotradas muy sofisticadas. Sin embargo, la puesta en práctica de estas soluciones es, y permanecerá, cuestionable hasta que se desarrollen soluciones que permitan garantizar niveles de confiabilidad aceptables en el uso de dichas soluciones.

Es cierto que hasta el momento no se ha estudiado con suficiente profundidad la viabilidad del uso de tecnologías orientadas a componentes para el desarrollo de sistemas empotrados confiables y seguros. Estas tecnologías han demostrado ya su validez para la producción de soluciones empotradas de tiempo real con arquitecturas flexibles, abiertas y evolutivas. Sin embargo, pocos han sido hasta el momento los esfuerzos centrados en estudiar hasta qué punto, y de qué manera, dichas arquitecturas permiten el despliegue en el sistema de estrategias de tolerancia a fallos tanto de tipo accidental como malicioso. En este sentido muchas preguntas restan sin respuesta: como por ejemplo cómo declinar una estrategia de tolerancia a fallos utilizando componentes; o a qué nivel del sistema (hardware o software) es más interesante (atendiendo a cuestiones de costes de desarrollo y mantenimiento) desarrollar dichos componentes, o cómo coordinar las actividades de los mismos, etc.

Así pues, la tolerancia a fallos es todavía un aspecto no funcional poco tratado, y por tanto, mal soportado por las arquitecturas empotradas basadas en componentes. Aún quedan muchas cuestiones por resolver también relativas a ¿Qué tipo de estrategias de tolerancia a fallos son aplicables a cada nivel de un sistema empotrado y como aplicarlas? ¿Que tipos de fallos se pueden tolerar? y yendo más allá, ¿En qué situaciones puede resultar interesante definir soluciones de tolerancia mixtas (hardware-software) en las que sea necesario coordinar la actividad de componentes a distintos niveles?

Es importante señalar que el desarrollo a base de componentes hace referencia de alguna manera al desarrollo de sistemas complejos a partir de otros sistemas más sencillos. De hecho, esta idea es recursiva. Dos componentes pueden definir un sistema, que puede ser visto, a un nivel de abstracción mayor, como un componente utilizable para definir otro sistema de mayor envergadura.

Por todo ello, se hace necesario para el futuro abordar la definición de estrategias de tolerancia a fallos bajo dos perspectivas, una mono-nivel (en la que se debería estudiar la viabilidad de las soluciones sobre componentes a nivel software y hardware) y otra multi-nivel (en la que se debería valorar la necesidad, los pros y los contras de definir soluciones de tolerancia a fallos que exijan la coordinación de la actividad de componentes implementados a distintos niveles de un sistema empotrado). Así pues, se evidencia la necesidad de proporcionar, además de soluciones de diseño, soluciones de implementación, que se traduzcan en herramientas capaces de automatizar, o al menos dar soporte a la implementación de aquellas interfaces que cada componente deba proporcionar para poder ser utilizados de manera confiable en el sistema.

Por tanto, como se puede ver desde la perspectiva del desarrollo de sistemas basados en componentes, aún queda mucho trabajo por realizar para llegar a conseguir que la integración de unos con otros haga que los sistemas sean cada día más confiables.

Apéndice A

NORMAS ISO, IEC, UNE

A1 INTRODUCCIÓN

ISO (Organización Internacional de Normalización) e **IEC** (Comisión Electrotécnica Internacional) constituyen el sistema especializado para la normalización a nivel mundial. Los organismos nacionales miembros de ISO o de IEC participan en el desarrollo de normas internacionales a través de los comités técnicos establecidos por las organizaciones respectivas para realizar acuerdos en los campos específicos de la actividad técnica. Los comités técnicos de ISO e IEC colaboran en los campos de interés mutuo. Otras organizaciones internacionales, gubernamentales y no gubernamentales, en colaboración con ISO e IEC, también toman parte en estos trabajos. Los proyectos de las normas internacionales se elaboran de acuerdo con las reglas contenidas en las Directivas de ISO/IEC, Parte 3.

En el campo de las tecnologías de la información, ISO e IEC han establecido un comité técnico conjunto denominado, ISO/IEC JTC l. Los borradores de normas internacionales adoptados por el comité conjunto circulan por los organismos nacionales para su voto. La publicación como norma internacional requiere la aprobación de al menos el 75% de los organismos nacionales con derecho a voto.

Dentro de los comités nacionales mencionar a AENOR por ser el representante de ISO en España.

AENOR es una entidad española, privada, independiente, sin ánimo de lucro, reconocida en los ámbitos nacional, comunitario e internacional, contribuye, mediante el desarrollo de las actividades de normalización y certificación (N+C), a mejorar la calidad en las empresas, sus productos y servicios, así como proteger el medio ambiente y, con ello, el bienestar de la sociedad.

Su compromiso es:

- Elaborar normas técnicas españolas con la participación abierta a todas las partes interesadas y colaborar impulsando la aportación española en la elaboración de normas europeas e internacionales.

- Certificar productos, servicios y empresas (sistemas) confiriendo a los mismos un valor competitivo diferencial que contribuya a favorecer los intercambios comerciales y la cooperación internacional.

- Orientar la gestión a la satisfacción de sus clientes y la participación activa de sus personas, con criterios de calidad total, y obtener resultados que garanticen un desarrollo competitivo.

- Impulsar la difusión de una cultura que les relacione con la calidad y les identifique como apoyo a quien busca la excelencia.

Fue designada para llevar a cabo estas actividades por la Orden del Ministerio de Industria y Energía, de 26 de febrero de 1986, de acuerdo con el Real Decreto 1614/1985 y reconocida como organismo de normalización y para actuar como entidad de certificación por el Real Decreto 2200/1995, en desarrollo de la Ley 21/1992, de Industria.

Su presencia en los foros internacionales, europeos y americanos garantiza la participación de España en el desarrollo de la normalización y el reconocimiento internacional de la certificación de AENOR.

A2 NORMA ISO/IEC 9126-1.

Ingeniería del software. Calidad del producto software. Modelo de calidad.

La Norma Internacional ISO/IEC 9126-1 fue elaborada por el SC 7, *Ingeniería del Software y Sistemas* del Comité Técnico Conjunto ISO/IEC JTC 1, *Tecnología de la Información.* La primera edición de la Norma ISO/IEC 9126-1, junto con las otras partes de la misma anula y sustituye a la Norma ISO/IEC 9126:1991, que ha sido técnicamente revisada.

La Norma ISO/IEC 9126 consta de las siguientes partes que comparten el mismo título general: Ingeniería del software. Calidad del producto.

- Parte 1: Modelo de calidad
- Parte 2: Métricas externas
- Parte 3: Métricas internas
- Parte 4: Métricas de calidad en uso

El anexo A (de la norma) es normativo y forma parte de esta parte de la Norma ISO/IEC 9126-1. Los anexos B, C y D (de la norma) tienen únicamente propósito informativo.

La Norma ISO/IEC 9126 (1991): Evaluación del producto software. Características de calidad y guías para su uso, fue desarrollada para dar apoyo a necesidades relacionadas con la calidad del software, define seis características de calidad y describe un modelo de proceso de evaluación del producto software.

Cuando las características y las métricas asociadas se puedan usar no sólo para evaluar el producto software, sino también para definir requisitos de calidad y otros usos, la Norma ISO/IEC (1991) ha sido sustituida por dos normas relacionadas que constan de varias partes: las Normas ISO/IEC 9126 (Calidad del producto software) e ISO/IEC 14598 (Evaluación del producto software). Las características de calidad del producto software que se definen en esta

parte de la Norma ISO/IEC 9126 se pueden usar para especificar requisitos funcionales y/o no funcionales tanto de cliente como de usuario.

Esta Norma ISO/IEC 9126 describe un modelo en dos partes para la calidad del producto software: a) calidad interna y externa, y b) calidad en uso. La primera parte del modelo especifica seis características para la calidad interna y externa, que se subdividen posteriormente en subcaracterísticas. Estas subcaracterísticas se manifiestan externamente cuando el software se usa como parte de un sistema informático, y son el resultado de los atributos internos del software. Esta parte de la Norma ISO/IEC 9126 no elabora el modelo de calidad interna y externa más allá del nivel de subcaracterística.

La segunda parte del modelo especifica cuatro características de calidad en uso, pero no elabora el modelo de calidad en uso más allá del nivel de característica. La calidad en uso es el efecto combinado para el usuario de las seis características de calidad del producto software.

Esta parte de la Norma ISO/IEC 9126 permite especificar y evaluar la calidad de los productos software desde diferentes perspectivas por parte de aquellos agentes involucrados con la adquisición, los requisitos, el desarrollo, el uso, la evaluación, el soporte, el mantenimiento, el aseguramiento de la calidad y la auditoria del software. Por ejemplo, se puede usar por los desarrolladores, compradores, personal de aseguramiento de la calidad, y evaluadores independientes, especialmente aquellos responsables de especificar y evaluar la calidad del producto software.

Ejemplos de usos del modelo de calidad definido en esta parte de la Norma ISO/IEC 9126 son:

- validar la completitud de una definición de requisitos;
- identificar los requisitos del software;
- identificar objetivos para el diseño del software;
- identificar objetivos para las pruebas del software;
- identificar los criterios de aceptación para un producto software completado.

A2.1 Marco de referencia del modelo de calidad

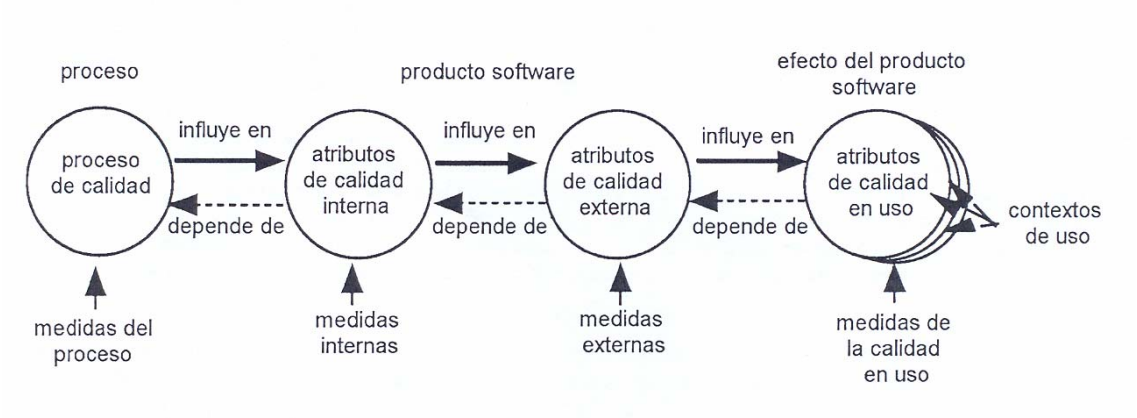


Figura A.1: Calidad en el ciclo de vida

La evaluación de los productos software, para que satisfagan las necesidades de calidad software, es uno de los procesos del ciclo de vida del desarrollo de software. La calidad del producto software se puede evaluar midiendo los **atributos internos** (normalmente medidas estáticas de productos intermedios), o midiendo los **atributos externos** (normalmente midiendo

el comportamiento del código en ejecución), o midiendo los atributos de la **calidad en uso**. El objetivo es que el producto tenga el efecto requerido en un contexto de uso particular.

La calidad del proceso (la calidad de cualquiera de los procesos definidos en la Norma ISO/IEC 12207) contribuye a mejorar la calidad del producto, y la calidad del producto contribuye a mejorar la calidad en uso. Así pues, la evaluación y mejora de un proceso es un mecanismo para mejorar la calidad del producto, y evaluar y mejorar la calidad del producto es un mecanismo para mejorar la calidad en uso. Igualmente, la evaluación de la calidad en uso puede proporcionar información para mejorar el producto, y la evaluación del producto puede proporcionar información para mejorar el proceso.

Así pues, la norma define como **calidad externa** a la totalidad de características del producto software desde una perspectiva externa. Ésta es la calidad cuando el software se ejecuta, y se mide y evalúa normalmente durante las pruebas en un entorno simulado con datos simulados y usando métricas externas. Durante las pruebas, se deberían encontrar y corregir muchos fallos. Sin embargo, algunos fallos pueden permanecer aun después de las pruebas. Dado que la arquitectura u otros aspectos fundamentales del diseño son difíciles de corregir, estos aspectos fundamentales del diseño normalmente permanecen sin cambios a lo largo de las pruebas.

A2.2 Uso de un modelo de calidad

La calidad de un producto software se debería evaluar usando un modelo de calidad definido. El modelo de calidad debería usarse a la hora de establecer los objetivos de calidad para los productos software y para los productos intermedios. La calidad del producto software debería descomponerse jerárquicamente en un modelo de calidad compuesto por características y subcaracterísticas que pueden usarse como una lista de comprobación de aspectos relacionados con la calidad.

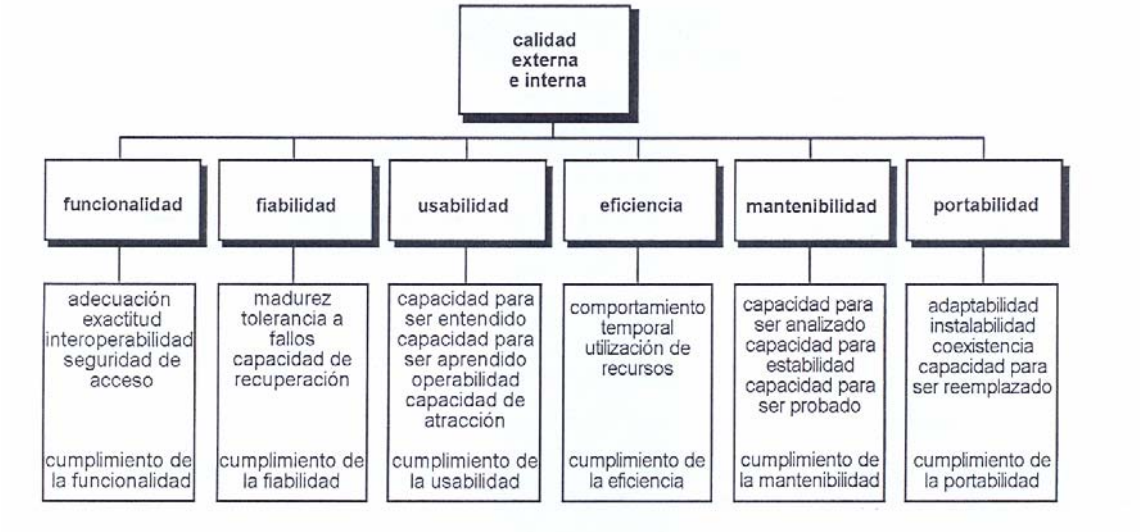


Figura A.2: Modelo para la calidad interna y externa propuesta por la norma

De este diagrama sólo se hará mención a aquellos aspectos que más interesan en relación con la presente tesis, así pues se define como:

Fiabilidad: La capacidad del producto software para mantener un nivel especificado de prestaciones cuando se usa bajo condiciones especificadas.

NOTA 1 - EL software no se degrada o envejece por sí solo. Las limitaciones en la fiabilidad se deben a defectos en requisitos, diseño e implantación. Los fallos debidos a este tipo de defectos dependen de la manera en que se usa el producto software y se seleccionan las opciones del programa, en lugar del tiempo transcurrido.

NOTA 2 - La definición de fiabilidad de la Norma ISO/IEC DIS 2382-14:1994 es: "La habilidad de una unidad funcional para llevar a cabo una función requerida ...". En este documento, la funcionalidad es sólo una de las características de la calidad del software. Así pues, la definición de fiabilidad se ha ampliado a "mantener un nivel de prestaciones especificado..." en lugar de " ... llevar a cabo una función requerida".

Madurez: La capacidad del producto software para evitar fallar como resultado de fallos en el software.

Tolerancia a fallos: La capacidad del producto software para mantener un nivel especificado de prestaciones en caso de fallos software o de infringir sus interfaces especificados.

NOTA - El nivel de prestaciones especificado podría incluir la capacidad de seguridad ante fallos.

Capacidad de recuperación: La capacidad del producto software para reestablecer un nivel de prestaciones especificado y de recuperar los datos directamente afectados en caso de fallo.

NOTA l - Tras un fallo, un producto software se encontrará no operativo durante un cierto periodo de tiempo cuya duración se evalúa mediante su capacidad de recuperación.

NOTA 2 - La disponibilidad es la capacidad del producto software de estar en un estado que permita llevar a cabo una función requerida en un momento determinado, bajo condiciones de uso especificadas. Externamente, se puede evaluar la disponibilidad mediante la relación entre el tiempo total y el tiempo en que el sistema está operativo. La disponibilidad es, así pues, una combinación de madurez, (que impacta en la frecuencia de fallo), la tolerancia a los fallos, y la capacidad de recuperación (que impacta en la duración del tiempo de caída tras cada fallo).

Mantenibilidad: La capacidad del producto software para ser modificado. Las modificaciones podrían incluir correcciones, mejoras o adaptación del software a cambios en el entorno, y requisitos y especificaciones funcionales.

Capacidad para ser analizado: La capacidad del producto software para ser diagnosticadas deficiencias o causas de los fallos en el software, o para identificar las partes que han de ser modificadas.

Estabilidad: La capacidad del producto software para evitar efectos inesperados debidos a modificaciones del software.

Capacidad para ser probado: La capacidad del producto software que permite que el software modificado sea validado.

Cumplimiento de la mantenibilidad: La capacidad del producto software para adherirse a normas o convenciones relacionadas con la mantenibilidad.

Portabilidad: La capacidad del producto software para ser transferido de un entorno a otro.

NOTA - El entorno puede ser organizativo, hardware o software.

A2.3 Métricas de Software

Dentro del apartado de métricas de Software destacar un texto que hace mención a las métricas externas y que justificaría la necesidad de utilización de herramientas de evaluación externa de la calidad del software.

Métricas externas: Las métricas externas usan medidas del producto software derivadas de medidas del comportamiento del sistema del que es parte, a través de probar, operar y observar el software ejecutable del sistema. En la Norma ISO/IEC 9126-2 se dan ejemplos de métricas externas. Las métricas externas proporcionan a los usuarios, evaluadores, probadores y desarrolladores el beneficio de que van a ser capaces de evaluar la calidad del producto software durante las pruebas o la operación.

A2.4 Otras normas de interés relacionadas:

IEC 60050-191	*Vocabulario Electrotécnico Internacional. Capítulo 191: Dependabilidad y calidad del servicio.*
IEEE 610.12-1990	*Glosario normalizado de terminología de ingeniería del software.*
ISO/IEC 2382-1:1993	*Tecnología de la Información. Vocabulario. Parte 1: Términos fundamentales.*
ISO/IEC 2382-14:1997	*Tecnología de la Información. Vocabulario. Parte 14: Fiabilidad, mantenibilidad y disponibilidad.*
ISO/IEC 2382-20:1990	*Tecnología de la Información. Vocabulario. Parte 20: Desarrollo de sistemas.*
ISO 8402: 1994	*Gestión de la calidad y aseguramiento de la calidad. Vocabulario.*
ISO 9001: 1994	*Sistemas de la calidad. Modelo para el aseguramiento de la calidad en el diseño, el desarrollo, la producción, la instalación y el servicio posventa.*
ISO/IEC TR 9126-2	*Ingeniería del Software. Calidad del producto. Parte 2: Métricas externas.*
ISO/IEC TR 9126-3	*Ingeniería del Software. Calidad del producto. Parte 3: Métricas internas.*
ISO/IEC TR 9126-4	*Ingeniería del Software. Calidad del producto. Parte 4: Calidad en el uso de métricas.*
ISO/IEC 12207:1995	*Tecnología de la información. Procesos del ciclo de vida del software.*
ISO/IEC 14598-2	*Tecnología de la Información. Evaluación del Producto Software. Parte 2: Planificación y gestión.*
ISO/IEC 14598-3	*Tecnología de la Información. Evaluación del Producto Software. Parte 3: Procedimiento para desarrolladores.*
ISO/IEC 14598-4:1999	*Tecnología de la Información. Evaluación del Producto Software. Parte 4: Procedimiento para compradores.*
ISO/IEC 14598-5:1998	*Tecnología de la Información. Evaluación del Producto Software. Parte 5: Procedimiento para evaluado res.*
ISO/IEC 14598-6	*Tecnología de la Información. Evaluación del Producto Software. Parte 6: Módulos de evaluación y documentación.*
ISO/IEC TR 15504 (todas las partes)	*Tecnología de la Información. Evaluación del proceso de software.*

A3 NORMA ISO/IEC 61508

Seguridad funcional de los sistemas eléctricos/electrónicos/electrónicos programables relacionados con la seguridad

Esta norma internacional trata de los aspectos a tener en consideración cuando se utilicen sistemas eléctricos/electrónicos/electrónicos programables (E/E/PE) para ejecutar funciones de seguridad.

Uno de los principales objetivos de esta norma internacional es permitir la elaboración de normas internacionales específicas a cada sector de aplicación por los comités técnicos responsables de los sectores correspondientes. Esto permite tener en cuenta el conjunto de los factores pertinentes para cada aplicación, y de responder a las necesidades específicas de cada uno de estos sectores.

Otro de los objetivos perseguidos por esta norma internacional es permitir el desarrollo de sistemas E/E/PE relacionados con la seguridad en ausencia eventual de normas internacionales para este sector de aplicación.

En particular esta norma trata de los sistemas E/E/PE relacionados con la seguridad en los que un fallo podría tener un impacto sobre la seguridad de las personas y/o el entorno considerable; sin embargo, se reconoce que los fallos pueden provocar serias consecuencias económicas, y en tales casos, esta norma también podría utilizarse para especificar cualquier sistema utilizado para la protección de un equipo o producto.

La norma esta constituida por siete partes (o documentos):

- La primera parte define los requisitos generales que se aplican a todas las otras partes que tratan temas más específicos.
- Las partes 2 y 3 proporcionan los requisitos suplementarios y específicos para los sistemas E/E/PE relacionados con la seguridad [para el hardware (soporte físico) y el software (soporte lógico)].
- La parte 4 proporciona las definiciones y abreviaturas que se utilizan a lo largo de esta norma.
- La parte 5 proporciona las directrices para la puesta en marcha de la determinación de los niveles de integridad de seguridad, definidos en la parte 1, presentando ejemplos de métodos.
- La parte 6 proporciona las directrices para la aplicación de las partes 2 y 3;
- La parte 7 contiene una presentación de técnicas y de medidas.

A continuación en la siguiente figura A.3, se puede ver cual es la estructura general de esta norma, así como las distintas partes que la componen.

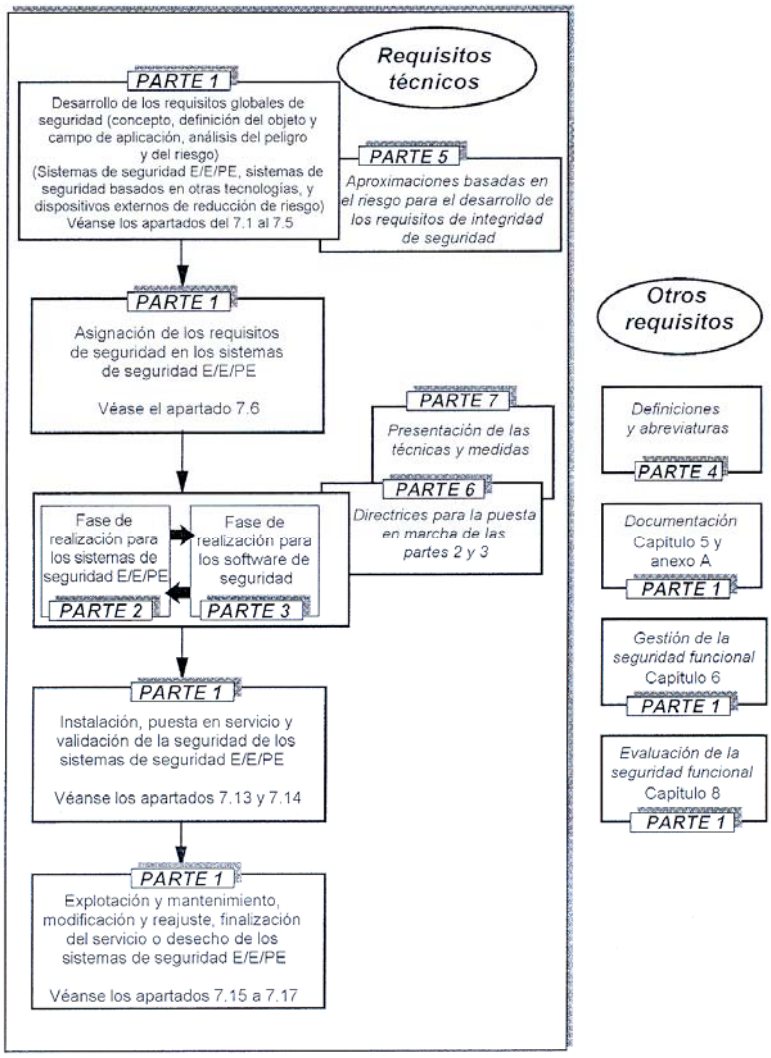


Figura A.3: Estructura general de la norma

A3.1 Parte 3: Requisitos del software (soporte lógico)

A continuación se estudian brevemente algunas particularidades de la norma que sirven para justificar el estudio de la misma en relación con la presente tesis. Nos centraremos en la parte tercera que habla explícitamente de los requisitos del software en cuanto a los sistemas de seguridad.

La parte tercera que habla de los requisitos del software y según se puede observar en la siguiente figura A.4, dentro de lo que sería el ciclo de vida de seguridad del software, se hace mención clara a una planificación y consiguiente validación de la seguridad del software (fases 9.2 y 9.6), y a una integración hardware/software de los elementos electrónicos programables (fase 9.4), lo cual justifica la necesidad de evaluar por un lado, que el sistema sea seguro y por otro, que el hardware y software se integren mutuamente bien para conseguir también esos requisitos de seguridad que el sistema debe proporcionar. De hecho se dice que hay que combinar el software y el hardware en la electrónica programable relacionada con la seguridad para asegurar su compatibilidad y respuesta a los requisitos del nivel de integridad de seguridad previsto.

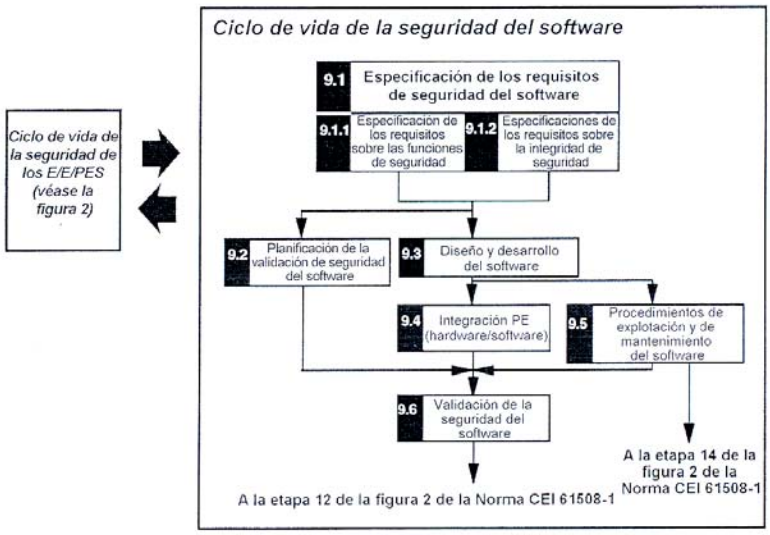


Figura A.4: Ciclo de vida de seguridad del SW (en fase de realización)

En la siguiente figura A.5 se puede observar como dentro del ciclo de vida del desarrollo del software se ve claro que cada fase se debe verificar y en última instancia se deben realizar unos ensayos de validación para las especificaciones de los requisitos de seguridad del software. En concreto se dice que en cuanto a la verificación del software: en los límites impuestos por el nivel de integridad de seguridad se deben probar y evaluar las salidas de una fase dada del ciclo de vida de la seguridad del software con el fin de verificar la conformidad y la coherencia en relación a las salidas y a las normas proporcionadas en las entradas de cada fase.

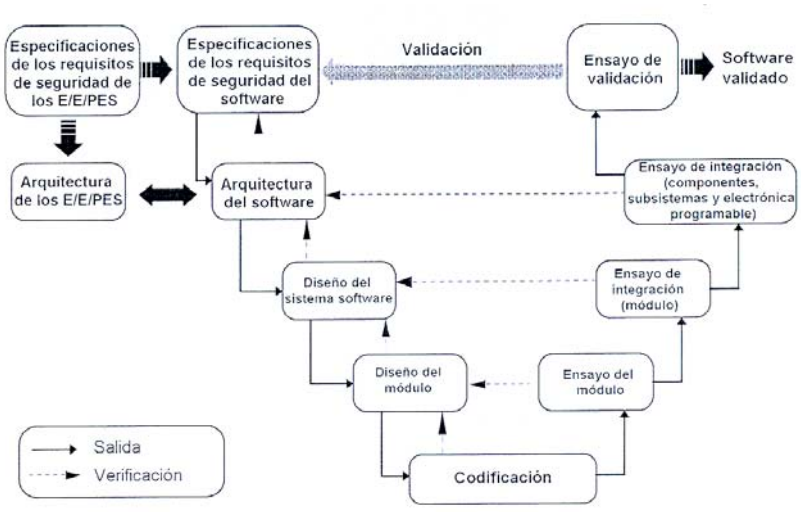


Figura A.5: Integridad de seguridad del SW y Ciclo de vida del desarrollo

Dentro de los requisitos que se establecen para la validación del software se dice que:

> ”*El ensayo debe ser el método principal de validación para el software: las actividades de validación se pueden completar con las operaciones de animación y modelización. Que el software se debe ejecutar simulando las señales de entrada presentes en la explotación normal, los sucesos previstos y las condiciones no deseadas requiriendo una acción del sistema*”.

En cuanto a la verificación del software además se dice que todas las interfaces de instalación y su software asociado (es decir, sensores y accionadores, interfaz fuera de línea, etc.) se deben verificar en lo referente a:

- Detección de las averías de la interfaz previstas.
- Tolerancia de las averías de la interfaz previstas.

Además, todas las interfaces de comunicación y su software asociado se debe verificar para asegurar que tienen un nivel adecuado de:

- Detección de averías.
- Protección contra la alteración.
- Validación de los datos.

Dentro de la selección de técnicas y medidas para evaluar los requisitos de seguridad del software (EN 61508-3:2001 Anexo A), en relación al ensayo e integración del software dentro de su diseño y desarrollo se propone como altamente recomendable los ensayos de caja negra, así mismo también para las pruebas de integración de la electrónica programable (hardware y software), y para los ensayos de validación de la seguridad del software es también altamente recomendable la técnica de ensayo probabilística.

En cuanto a la verificación del software comentar que para llevar a cabo la misma, se propone la realización de análisis dinámicos y estáticos del software así como ensayos de integración de los diferentes módulos que pueden componer éste y la electrónica programable entorno a sus especificaciones de seguridad.

Por último reseñar que en la evaluación de la seguridad funcional del software se propone realizar un análisis de las averías, análisis de las averías de causa común de un software diversificado y diagramas de bloques de fiabilidad.

A3.2 Otras consideraciones

Dentro de las definiciones y abreviaturas de la parte 4 de la norma, se definen los conceptos de anomalía, fallo, error, tolerancia a fallos, etc. En este caso se va a estudiar la de fallo sistemático por ser el tipo de fallos relacionados con el diseño del software. Así, "*fallo sistemático***"** es: Fallo relacionado de forma determinista a una causa concreta, que sólo se puede eliminar por modificación del diseño o del proceso de fabricación, procedimientos de funcionamiento, documentación u otros factores correspondientes.

En la parte 5 se habla de la reducción de riesgo. La reducción de riesgo necesaria (véase el apartado 3.5.14 de la Norma CEI 61508-4) es la reducción de riesgo que se debe realizar para alcanzar el riesgo tolerable en una situación específica. El concepto de reducción de riesgo necesaria es de una importancia fundamental en la realización de las especificaciones de requisitos de seguridad para los sistemas E/E/PE relacionados con la seguridad (en particular, los requisitos de integridad de seguridad que forman parte de la especificación de los requisitos de seguridad). La determinación del riesgo tolerable por un acontecimiento peligroso tiene por objeto considerar la frecuencia (o probabilidad) del acontecimiento peligroso y sus consecuencias específicas. Los sistemas relacionados con la seguridad están diseñados para reducir la frecuencia (o probabilidad) del acontecimiento peligroso y/o las consecuencias de dicho acontecimiento.

La integridad de seguridad se define como la probabilidad para que un sistema relacionado con la seguridad ejecute de forma satisfactoria las funciones de seguridad requeridas, en todas las condiciones especificadas y en un periodo de tiempo especificado (apartado 3.5.2 de la Norma CEI 61508-4), como se puede observar siempre respetando las restricciones temporales.

Se considera que la integridad de seguridad se compone por los dos siguientes elementos:

- Integridad de seguridad del **hardware**: esta parte de la integridad de seguridad está relacionada con los fallos aleatorios del hardware en un modo de fallo peligroso (véase el apartado 3.5.5 de la Norma CEI 61508-4).

- Integridad de seguridad **sistemática**; esta parte de la integridad de seguridad está relacionada con los fallos sistemáticos en modo de fallo peligroso (véase el apartado 3.5.4 de la Norma CEI 61508-4). Aunque la tasa media de fallo debida a los fallos sistemáticos pueda estimarse, los datos de fallo resultantes de una anomalía de diseño o de fallos de causa común hacen que pueda ser difícil prever la distribución de los fallos. La incertidumbre de los cálculos de probabilidad de fallo para una situación específica aumenta. Entonces, es necesario optar por las mejores técnicas para minimizar esta incertidumbre. De todos modos, las medidas tomadas para reducir la probabilidad de fallo aleatorio del hardware no tienen necesariamente un efecto correspondiente sobre la probabilidad de fallo sistemático. Las técnicas tales como la redundancia del hardware por dos canales, que son muy eficaces para controlar los fallos aleatorios del hardware, tienen un efecto muy limitado en la reducción de fallos sistemáticos.

La integridad de seguridad requerida de un sistema E/E/PE relacionado con la seguridad, o de sistemas relacionados con la seguridad, basados en otras tecnologías y dispositivos externos de reducción de riesgo, debe estar a un nivel tal que:

- la frecuencia de fallo del sistema relacionado con la seguridad sea suficientemente baja como para evitar que la frecuencia de los acontecimientos peligrosos no exceda el valor requerido para alcanzar el riesgo tolerable, y/o

- los sistemas relacionados con la seguridad tengan una acción suficiente sobre las consecuencias de los fallos para alcanzar un nivel de riesgo tolerable.

A continuación se verán algunos gráficos de interés relacionados con la degradación en el concepto de riesgo tolerable.

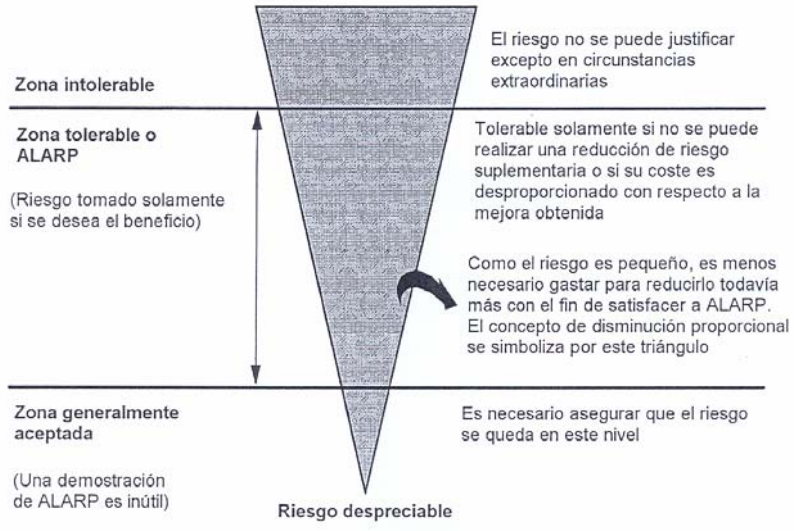


Figura A.6: Riesgo tolerable y ALARP (As Low As Reasonably Practicable)

Tabla A.1: Ejemplo de la clasificación de los accidentes en función de los riesgos

Frecuencia	Consecuencia			
	Catastrófica	Crítica	Marginal	Despreciable
Frecuente	I	I	I	II
Probable	I	I	II	III
Ocasional	I	II	III	III
Poco frecuente	II	III	III	IV
Improbable	III	III	IV	IV
Increíble	IV	IV	IV	IV

NOTA 1 – La atribución real de las clases de riesgo I, II, III, IV dependen del sector de aplicación e igualmente de las frecuencias reales (frecuente, probable, etc.). En consecuencia, conviene que esta tabla sea percibida como un ejemplo de cómo una tabla se puede enriquecer, mejor que una especificación para un uso futuro.

NOTA 2 – Una percepción de la determinación de los niveles de integridad de seguridad, a partir de las frecuencias presentadas en esta tabla, se encuentra en el anexo C.

Tabla A.2: Interpretación de las clases de riesgo

Clase de riesgo	Interpretación
Clase I	Riesgo intolerable
Clase II	Riesgo indeseable, tolerable únicamente si es imposible reducir el riesgo o el coste de la reducción es desproporcionado con relación a la mejora conseguida
Clase III	Riesgo tolerable si el coste de la reducción del riesgo es superior a la mejora conseguida
Clase IV	Riesgo despreciable

En la parte 7 de la norma se proponen técnicas y medidas. A continuación se explican algunas que parecen interesantes y que son más representativas de la presente tesis doctoral.

- **ENSAYO DE "CAJA NEGRA"**

  Objetivo: Controlar el comportamiento dinámico en las condiciones reales de funcionamiento. Revelar los fallos para que la especificación funcional se satisfaga y se evalúe su utilización y su robustez.

  Descripción: Las funciones de un sistema o de un programa se ejecutan en un entorno específico con los datos de ensayo especificados, deducidos sistemáticamente de la especificación relativa a los criterios establecidos. Esto pone en evidencia el comportamiento del sistema y permite una comparación con la especificación. Ningún conocimiento de la estructura interna del sistema se utiliza para dirigir el ensayo. El objetivo es determinar si la unidad funcional realiza correctamente todas las funciones requeridas por la especificación. La técnica que consiste en formar dos clases de equivalencia es un ejemplo de los criterios para definir los datos de ensayo "caja negra". El conjunto de datos de entrada se subdivide en rangos de valores de entrada "clases de equivalencia" con la ayuda de la especificación. Los casos de ensayo se forman a partir de:

  - los datos en los rangos permitidos;
  - los datos fuera de los rangos permitidos;
  - los datos en los límites de los rangos;
  - los valores extremos;
  - las combinaciones de las clases anteriores.

Otros criterios específicos pueden ser eficaces para elegir el caso de ensayo en las diferentes actividades de ensayo (ensayo del módulo, ensayo de integración y ensayo del sistema). Por ejemplo, para los ensayos del sistema en el marco de una validación, se utiliza el criterio "condiciones de explotación extremas".

- **ENSAYO "ESTADÍSTICO"**

  Objetivo: Verificar el comportamiento dinámico del sistema relacionado con la seguridad y evaluar su utilización y su robustez.

  Descripción: Esta aproximación ensaya un sistema o un programa con unos datos de entrada elegidos en función de la distribución estadística esperada de los datos de entrada reales, es decir el perfil operacional.

- **ANÁLISIS DE FALLOS**

  Objetivo: Analizar un diseño del sistema estudiando todas las fuentes posibles de fallos de los componentes de este sistema determinando los efectos de estos fallos en el comportamiento y la seguridad del sistema.

  Descripción: El análisis se hace generalmente durante una reunión de ingenieros. Los componentes de un sistema se analizan uno después de otro para establecer un conjunto de modos de fallo del componente, sus causas y sus efectos, así como los procedimientos y las recomendaciones para la detección. Si se tienen en cuenta las recomendaciones, se documentan como las acciones de corrección que se han tomado.

- **ANÁLISIS DE LOS MODOS DE FALLO, DE SUS EFECTOS Y DE SU CRITICIDAD**

  Objetivo: Establecer un orden de criticidad de los componentes que puedan ser el origen de un daño, un perjuicio o una degradación del sistema a través de fallos únicos, con el fin de determinar que componentes pueden necesitar de una atención particular y de medidas de supervisión durante el diseño o la explotación.

  Descripción: La criticidad se puede clasificar de varias formas. El método más laborioso se describe por la *Society for Automotive Engineers* (SAE) en la Norma ARP 926. En este procesamiento, el grado de criticidad de un componente se indica por el número de fallos de un tipo particular esperado durante el transcurso de un ciclo de un millón de operaciones produciéndose en un modo crítico. El grado de criticidad es una función de nuevos parámetros, en la que la mayor parte se deben medir. Un método muy simple para determinar la criticidad es multiplicar la probabilidad de fallo del componente por el daño que se podría causar; este método es similar a la evaluación simple del factor de riesgo.

- **ENSAYO "PROBABILÍSTICO"**

  Objetivo: Obtener una indicación cuantitativa relativa a las propiedades de confianza del software examinado.

  Descripción: Esta indicación cuantitativa puede tener en cuenta los niveles de fiabilidad y de significación asociados y puede corresponder a:

  - una probabilidad de fallo por demanda;
  - una probabilidad de fallo durante un cierto periodo de tiempo; y
  - una probabilidad de confinamiento de los errores.

A partir de estas indicaciones, se pueden obtener otros parámetros, como:

- la probabilidad de ejecución sin fallo;
- la probabilidad de supervivencia;
- la disponibilidad;
- el MTBF o la tasa de fallo; y
- la probabilidad de ejecución segura.

Las consideraciones probabilísticas se basan o bien en el ensayo probabilístico o bien en la experiencia de explotación. Normalmente, el número de casos de ensayo o de casos observados en explotación es muy importante. Las herramientas de ensayo automatizadas se utilizan habitualmente para proporcionar los datos de ensayo y supervisar las salidas de ensayo. La utilización de bancos de ensayos del software, la ejecución y la supervisión de ensayos individuales se determinan en función de los objetivos detallados de los ensayos, como el descrito anteriormente.

- **ENSAYO "DE INTERFAZ"**

  Objetivo: Detectar los errores en las interfaces de los subprogramas.

  Descripción: Son posibles varios niveles de detalle o de completitud de los ensayos. Los niveles más importantes son los ensayos que ponen en juego:

  - todas las variables de interfaz en sus valores extremos
  - todas las variedades de interfaz consideradas individualmente en sus valores extremos con las otras variables de interfaz en su valor normal;
  - todos los valores del dominio de cada variable de interfaz con las otras variables de interfaz en su valor normal;
  - todos los valores de todas las variables combinadas (esto sólo se puede realizar para pequeñas interfaces);
  - las condiciones de ensayos especificadas relativas a cada llamada de cada subprograma.

  Estos ensayos son particularmente importantes cuando las interfaces no contienen reafirmaciones después de la elaboración de nuevas configuraciones de subprogramas preexistentes.

- **ANÁLISIS DE LOS VALORES LÍMITES**

  Objetivo: Detectar los errores lógicos en las fronteras o límites de los parámetros.

  Descripción: El dominio de entrada del programa se divide en un cierto número de clases de entrada después de la relación de equivalencia. Conviene que los ensayos cubran las especificaciones coincidiendo con las del programa. La utilización del valor cero en el caso de una traducción directa así como indirecta a menudo está sujeta a errores y necesita una atención especial:

  - dividir por cero;
  - caracteres ASCII blancos;
  - elemento de la lista o pila vacía;
  - matriz llena;
  - entrada de la tabla en cero.

  Normalmente, los límites para las entradas corresponden directamente con las del rango de salida. Conviene que los casos de ensayo se describan de forma que fuerce la salida para

que alcance sus valores límites. También es necesario considerar si es posible o no especificar un caso de ensayo para aquel cuya salida supere los valores límites de la especificación.

A3.3 Otras normas de interés relacionadas:

UNE-EN 60300-1:1996	Gestión de la Confiabilidad. Parte 1: Gestión del programa de Confiabilidad.
UNE-EN 60300-1:2004	Gestión de la Confiabilidad. Parte 1: Sistemas de gestión de la Confiabilidad.
UNE-EN 60300-2:1997	Gestión de la Confiabilidad. Parte 2: Elementos y tareas del programa de Confiabilidad.
CEI 60300-1:1993	Gestión de la Confiabilidad. Parte 1: Gestión del programa de Confiabilidad.
CEI 60300-1:2003	Gestión de la Confiabilidad. Parte 1: Sistemas de gestión de la Confiabilidad.
CEI 60300-2:1995	Gestión de la Confiabilidad. Parte 2: Elementos y tareas del programa de Confiabilidad.
CEI 60300-3-11:1999	Gestión de la Confiabilidad. Parte 3-11: Guía de aplicación. Mantenimiento centrado en la fiabilidad.
CEI 60300-3-2:1993	Gestión de la Confiabilidad. Parte 3: Guía de aplicación. Sección 2: Recogida de datos de Confiabilidad en la explotación.
CEI 60300-3-4:1996	Gestión de la Confiabilidad. Parte 3: Guía de aplicación. Sección 4: Guía para la especificación de los requisitos de Confiabilidad.
CEI 60300-3-5:2001	Gestión de la Confiabilidad. Parte 3-5: Guía de aplicación. Condiciones para los ensayos de fiabilidad y principios para la realización de contrastes estadísticos.
CEI 60300-3-7:1999	Gestión de la Confiabilidad. Parte 3-7: Guía de aplicación. Cribado de fiabilidad mediante esfuerzos del hardware electrónico.
CEI 60300-3-9:1995	Gestión de la Confiabilidad. Parte 3: Guía de aplicación. Sección 9: Análisis del riesgo de sistemas tecnológicos.
UNE 200001-3-11:2003	Gestión de la Confiabilidad. Parte 3-11: Guía de aplicación. Mantenimiento centrado en la fiabilidad.
UNE 200001-3-1:1998	Gestión de la Confiabilidad. Parte 3: Guía de aplicación. Sección 1: Técnicas de análisis de la Confiabilidad: Guía metodológica.
UNE 200001-3-2:2001	Gestión de la Confiabilidad. Parte 3: Guía de aplicación. Sección 2: Recogida de datos de Confiabilidad en la explotación.
UNE 200001-3-4:1999	Gestión de la Confiabilidad. Parte 3: Guía de aplicación. Sección 4: Guía para la especificación de los requisitos de Confiabilidad.
UNE 200001-3-5:2002	Gestión de la Confiabilidad. Parte 3-5: Guía de aplicación. Condiciones para los ensayos de fiabilidad y principios para la realización de contrastes estadísticos.

Referencias

[Aidernark0l] J. Aidernark, J. Vinter, P. Folkesson, J. Karlsson. "*GOOFI: Generic Object-Oriented Fault Injection tool*". In Proc. DSN 2001, Goteborg, Suecia, 2001.

[Aidernark03] J. Aidernark, J. Vinter, P. Folkesson, J. Karlsson. "*GOOFI: Generic Object-Oriented Fault Injection tool*". In Proc. DSN 2003, pp.668. San Francisco, EEUU, 2003.

[Arlat90] J. Arlat, M. Aguera, L. Arnat, 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, no. 2, vol. 16, pp. 166-182, 1990.

[Ashling03] Ashling "*VITRA Ernulator*", www.ashling.com, DS224 V3. 2003.

[Avizienis04] Avizienis, A., Laprie, J.C., Randell, B. and Landwehr, C., "*Basic Concepts and Taxonomy of Dependable and Secure Computing*". IEEE transactions on dependable and secure computing, vol. 1, no. 1, pp. 11-33, January-March 2004.

[Aylor92] J.H. Aylor, R. Waxrnan, B.W. Johnson, R.D. Williarns. "*The Integration of Performance and Functional Modelling III VHDL*". Performance and Fault Modelling with VHDL. , pp. 22-145. Prentice Hall, 1992.

[Barbosa05] Barbosa, R., et al. "*Assembly-Level Pre-Injection analysis for improving fault injection efficiency*". Lecture notes on computer Sciences LNCS 3463. Dependable Computing EDCC-5 Conference. April 2005. Budapest, Hungary.

[Basili01] Basili, V. R. y Boehm, B. (2001). "*COTS-Based Systems Top 10 List*". IEEE Computer, 30(5):91 93.

[Benso99a] A. Benso, M. Rebaudengo, M. Sonza Reorda. "*FlexFi: a flexible Fault Injection environment for microprocessor-based systems*". SAFECOMP 1999: l8th International Conference on Computer Safety, Reliability and Security, (Lecture Notes in Computer Science, Springer Verlag, A. Pasquini (Ed.)), pp. 323-335.

[Benso99b] A Benso, P.L. Civera, M. Rebaudengo, M. Sonza Reorda. "*A Low-Cost Programmable Board for Speeding-Up Fault Injection in Microprocessor Based Systems*". RÁMS99: Annual Reliability and Maintainability Symposium, Washington, DC (USA), January 1999, pp. 171-177.

[Berger03] A. Berger, M. Barr, "*Introduction to On-Chip Debug*", Embedded Systems Programming, Marzo 2003, pp. 47-48.

[Boue98] J. Boue, P. Pétillon, Y. Crouzet, "*Mephisto-L: A VHDL-based fault injection tool for the experimental assessment of fault tolerance*". In Proc. 28t1 International Symposium on Fault Tolerant Computing (FTCS-28), pp. 168-173, IEEE 1998.

[Bouricius69] W. Bouricius, W. Carter, P. Schneider, "*Reliability modelling techniques for self-repairing computer systems*", en Proceedings 24th ACM National Conference, pp. 295-309, 1969.

[Brown98] Brown, A. W. y Wallnau, K. C. (1998). "*The Current State of CBSE*". IEEE Software, 15(5):37 46.

[Campelo99a] J.C. Campelo, F. Rodriguez, P.J. Gil, J.J. Serrano. "*Design and Validation of a Distributed Industrial Control System's Nodes*". In Proc. 18th IEEE Symposium on Reliable Distributed Systems, pp. 300-30 1, 1999.

[Campelo99b] J.C. Campelo. "*Diseño y validación de nodos de proceso tolerantes a fallos en sistemas industriales distribuidos*", Tesis Doctoral, Departamento de Informática de Sistemas y Computadores, Universidad Politécnica de Valencia, 1999.

[Campelo99c] J.C. Campelo, P. Yuste, F. Rodriguez, P.J. Gil, J.J. Serrano. "*Dependability Evaluation of Fault Tolerant Distributed Industrial Control Systems*". En Lecture Notes in Computer Science (1586): Parallel and Distributed Processing. 11 IPPS/SPDP'99 Workshops, San Juan, PR (EEUU), pp. 384-388. Abril 1999.

[Campelo99d] J.C. Campelo, P. Yuste, F. Rodriguez, P.J. Gil, J.J. Serrano. "*Hierarchical Reliability and Safety models of Fault tolerant Distributed Industrial Control Systems*". En Lecture Notes in Computer Science (1698): Computer Safety, Reliability and Security (SAFECOMP'99), Toulouse (Francia), pp. 202-215. Septiembre, 1999.

[Campelo00] J.C. Campelo, P. Yuste, P.J. Gil, J.J. Serrano. "*DICOS: a real-time distributed industrial control system for embedded applications*". En 6th IFAC Workshop on Algorithms and Architectures for Real-Time Control (AARTC'2000), Palma de Mallorca (España) pp. 25-30. Mayo 2000.

[Campelo0l] J.C. Campelo, P. Yuste, P.J. Gil, J.J. Serrano. "*DICOS: a real-time distributed control system for embedded applications*". Control Engineering Practice, ISSN 0967-066 1, 9(4): 439-447, Abril 2001.

[Carney00] Carney, D. y Long, F. (2000). "*What do you mean by COTS'? Finally, a usefull answer*". IEEE Software, 17(2):83 86.

[Carreira95] J. Carreira, H. Madeira, J.G. Silva. "*Xception: Software fault injection and monitoring in processor functional units*". In Proc. DCCA-5, pp. 135-149, Urbana-Champaign, USA, Springer-Verlag, 1995.

[Carreira98] J. Carreira, H. Madeira, J.G. Silva. "*Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers*". IEEE Transactions on Software Engineering, vol. 24, 1998. pp. 125-136.

[Carrión03] C. Carrión, P. Yuste, L. Lernus, P. Gil. "*Sistema de Instrumentación para analizar la inyección de fallos en aplicaciones implementadas en Systems on Chip*". En Seminario Anual de Automática, Electrónica Industrial e Instrumentación (SAAEI), Vigo (España), Septiembre 2003.

[Chevochot01] P. Chevochot, L. Puaut. "*Experimental Evaluation of the Fail-Silent Behaviour of a Distributed Real-Time Run-Time Support built from COTS Components*", in Proc. DSN 2001, Goteborg, Sweden, 2001.

[CEC91] "*Information Technology Security Evaluation Criteria (ITSEC), Provisional Harmonised Criteria, Version 1.2*", Commision of the European Communities (CEC). 1991.

[Courtois92] B. Courtois, M.C. Gaudel, J.C. Laprie, D. Powell, "*Süreté de Fonctionnement Informatique. Evolutions 1987-1992. Tendances et Perspectives*", Rapport rédigé á la demande de la Direction Centrale de la Qualité du CNES, Diciembre 1992.

[Cunha99] J.C. Cunha, M.Z. Rela, J.G. Silva. "*Can Software Implemented Fault Injection be Used on Real-time systems?*". In Proc. 3rd European Dependable Computing Conference (EDCC-3), Prague, Czech Republic, pp. 209-226, 1999.

[Cunha02] J.C. Cunha, A. Correia, J. Henriques, M.Z. Rela, J.G. Silva. "*Reset-Driven Fault Tolerance*". European Dependable Computing Conference, EDCC4, pp. 102-120. Springer-Verlag LCNS 2485.

[Cygnal03] CYGNAL Integrated Products, Inc "*Mixed-Signal 32KB ISP FLASH MCU Family*",2002, pp.l62, l68.

[Damm88] A. Damm. "*Experimental Evaluation of Error-Detection and selfchecking Coverage of Components of a distributed Real-Time System*". Doctoral Dissertation, Technical University of Vienna, 1988.

[DBench04] "*DBench Dependability Benchmarks*". Deliverables and final report of the DBench Project, IST 2000-25425. Available online at: http://www.laas.fr/dbench.

[De Andrés03] D. De Andrés, J. Albaladejo, P. Yuste, L. Lemus, P. Gil. "*Fault Tolerant Vision Subsystem for a Mobile Platform*". In Proceedings of 4th Workshop on European Scientific and Industrial Collaboration. Vol. 1 (WESIC 2003), Miskolc (Hungría), pp. 97-104. Mayo 2003.

[Dean97] Dean, J. y Vigdcr, M. (1997). "*System Implementation Using Commercial Off-The-Shelf (COTS) Software*". En 9th Annual Software Terhnologg Conferenre (STC97), Salt Lake City, Utah, USA. http://seg.iit.nrc.ca/English/abstracts/ NRC4O 173. html.

[Delphi02] Delphi Technologies, Inc. "*ETC 2.1 Medium/Heavy Duty Diesel Controller*" 2002. USA.

[Dewey92] A. Dewey, A.J.D. Geus. "*VHDL: Toward a Unified View of Design*" IEEE Design and Test of Computers, pp 8-17, 1992.

[DOT/FAA02] DOT/FAA. "*Study of Commercial Off-The-Shelf (COTS) Real-Time Operating Systems (RTOS) in Aviation Applications*". Technical Report DOT/FAA/AR-02/118, December 2002.

[Dugan89] 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.

[Duraes03] J. Duraes, H. Madeira. "*Definition of Software Fault Emulation Operators: a Field Data Study*". Proceedings of the 2003 International Conference on Dependable Systems and Networks, DSN03, pp. 105-114. 2003. San Francisco, USA.

[EABI98] Microcontroller Applications, IBM Microelectronics "*Developing PowerPC Embedded Application Binary Interface (EABI)*". *Programming PowerPC Embedded Processors Application Note.* September 21, 1998.

[Echte92] K. Echte, M. Leu, "*The EFA fault injector for fault tolerant distributed system testing*". IEEE Workshop on Fault Tolerant Parallel and Distributed Systems, pp. 28-3 5, Amherst, USA, Julio 1992.

[Embedded03] "*Nexus 5001 updates debug standard*", Embedded Systems Europe, Junio 2003, p7.

[ETAS97] ETAS GmbH. "*White Paper Application and Measuring Systems*". Version 1.0. 1997 Stuttgart.

[ETAS02] ETAS GmbH. "*ETKS2.0 Emulator Probe for Serial Debug Interfaces Data Sheet*". 2002 Stuttgart.

[ETAS03]ETAS GmbH. "*ETK. The Emulator Test Probe*". http://www.etas.info/html/products/am/etk/enproductsametkindex.php. Stuttgart, 2003.

[Fabre00] J.C. Fabre, M. Rodríguez, J. Arlat, F. Salles, J.M. Sizun. "*Building Dependable COTS Microkernel-based Systems using MAFALDA*". In Proc. 2000 Pacific Rim International Symposium on Dependable Computing (PRDC2000), Los Angeles, CA (USA), pp. 85-92, 2000.

[Folkesson98] P. Folkesson, 5. Svensson, J. Karlsson. "*A Comparison of simulation based and scan chain implementation fault injection*". Proc. 28th International Symposium on Fault Tolerant Computing (FTCS-28), pp. 284-293, IEEE 1998.

[Fuchs96] E. Fuchs. "*An Evaluation of the Error Detection Mechanisms in MARS using Software-Implemented Fault Injection*". 2nd European Dependable Computing Conference (EDCC-2), pp. 73-90. Taormina, Italy, EU. Octubre 1996.

[Gil92] 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.

[Gil96] P.J. Gil. "*Garantía de funcionamiento: Conceptos básicos y terminología*", Informe interno, Departamento de Ingeniería de Sistemas, Computadores y Automática (DTSCA), Universidad Politécnica de Valencia, 1996.

[Gil97a] P. Gil, J.C. Baraza, D. Gil, J.J. Serrano. "*High speed fault injector for safety validation of industrial machinery*". In Proc. 8th European Workshop of Dependable Computing (EWDC-8): Experimental validation of dependable systems, 1997.

[Gil97b] 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*". In Proc 6th IEEE International Conference on Advanced Computing (ADCOMP 97), pp. 466-474, 1997.

[Gil98] D.Gil, J.V. Busquets, J.C. Baraza, P.J. Gil. "*A fault injection tool for VHDL models*" Fast Abstracts 28th International Symposium on Fault Tolerant Computing (FTCS-28), pp. 72-73, IEEE 1998.

[Gil02] P. Gil, J. Arlat, H. Madeira, Y. Crouzet, T. Jarboui, K. Kanoun, T. Marteau, J. Duríes, M. Vieira, D. Gil, J.C. Baraza, J. Gracia. "*Fault Representativeness Deliverable" (ETIE2)* of

the European Project Dependability Benchmarking Dbench (IST-2000-25425) funded by the European Community under the "Information Society Technologies" Programme (1998-2002)". LAAS-CNRS Toulouse, France 2002.

[Gil06] P. Gil. "*Computación Confiable y Segura: Conceptos Básicos y Taxonomía*". Informe interno, DISCA. Adaptación artículo de Avizienis, A., Laprie, J.C., Randell, B. and Landwehr, C., "*Basic Concepts and Taxonomy of Dependable and Secure Computing*". IEEE transactions on dependable and secure computing, vol. 1, no. 1, pp. 11-33, January-March 2004. Valencia septiembre 2006.

[Gunneflo89] U. Gunneflo, J. Karlsson, J. Torin. "*Evaluation of error detection schemes using fault injection by heavy-ion radiation*". In Proc. 19th International Symposium on Fault Tolerant Computing (FTCS-19), pp. 340-347, 1989.

[Han95] 5. Han, K.G. Shin, H.A. Rosenberg. "*DOCTOR: An IntegrateD Software Fault InjeCTiOn EnviRonment for Distributed Real-Time Systems*". In Proc. IEEE mt. Symp. Computer Performance and Dependability, 1995, pp. 204-2 13.

[Hitex03] "*www.hitex.de/products.html"*

[Hsueh97] M. Sueh, T. Tsai, R.K. Iyer, "*Fault Injection Techniques and Tools*", IEEE Computer, 30(4):75-82, Abril 1997.

[IEEE97] "Colloquium on COTS and Safety Critical Systems".UK: Inst. of Electrical Eng. Computing and Control Division, 1997.

[Iribarne03] Luis F. Iribarne. "*Un Modelo de Mediación para el Desarrollo de Software basado en Componentes COTS*". Tesis Doctoral. Departamento de Lenguajes y Computación. Universidad de Almería. Dirigida por Dr. D. Antonio Vallecillo Moreno y Dr. D. José María Troya Linero. Julio, 2003.

[ISO/IEC 9126] "Ingeniería del software. Calidad del producto software. Modelo de calidad". International Standards Organization/International Electrochemical Commision. AENOR, Comité técnico AEN/CTN 71 Tecnología de la Información. Diciembre, 2004.

[ISO/IEC 61508] "Seguridad funcional de los sistemas eléctricos/electrónicos/electrónicos programables relacionados con la seguridad". International Standards Organization/International Electrochemical Commision. AENOR, Comité técnico AEN/CTN 200 normas básicas eléctricas. Marzo, 2003.

[Iyer95] R.K. Iyer. "*Experimental Evaluation*". Special Issue FTCS-25 Silver Jubilee, 25th IEEE Symp. on Fault Tolerant Computing, FTCS-25, pp. 115-132, Jun. 1995.

[Janz01] Janz W., Stehle J. "*OSEK real-time operating system. osCAN User manual*". Vector Technische Informatik.

[Kanawati95] G.A. Kanawati, N.A. Kanawati, J.A. Abraham. "*A Flexible Software Based Fault and Error Injection System*". IEEE transactions on Computers, Vol. 45, No. 2, 1995, pp. 248-259

[Kanoun89] K. Kanoun. "*Croissance de la Sürété de fonctionement des logiciels. Caracterisation — Modelisation — Evaluation*", Thése présentée a L'Institut National Polytechnique de Toulouse, Septiembre 1989.

[Kanoun05] Kanoun, K. et al. "*DBench - Dependability Benchmarking*", in Proc. of the Lecture Notes in Computer Science (LNCS), Springer-Verlag, 5th European Dependable Computing Conference (EDCC-5), April 2005, Budapest, Hungary.

[Kao93] 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, SE-19(1 1), pp. 1105-1118. Noviembre 1993.

[Kao94] W. Kao, R.K. Iyer. "*DEFINE: a distributed fault injection and monitoring environment*". Workshop on fault tolerant parallel and distributed systems. Junio 1994.

[Karlsson95] J.Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, J. Reisinger. "*Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture*". In Proc. 5111 International Working Conference on Dependable Computing for Critical Applications (DCCA-5), 1995.

[Koopman02] P. Koopman. "*What's wrong with Faul Injection as a Benchmarking Tool?*". DSN Workshop on Dependability Benchmarking, 2002.

[Kopetz97] H. Kopetz. "*Real-Time Systems: Design Principles for Distributed Embedded Applications*". Kluwer Academic Publishers, 1997. ISBN 0-7923-9894-7.

[Kropp98] N. Kropp, P. J. Koopman, and D. P. Siewiorek, "*Automated Robustness Testing of Off-the-Shelf Software Components*", in Proc. of the 28th IEEE Int. Symposium on Fault Tolerant Computing, pp. 230–239, Munich, Germany, 1998.

[Labrosse01] Jean J. Labrosse, *"MicroC/OS-II, The Real Time Kernel"*, Edit. R&D Books. USA, 2001. ISBN: 0-87930-543-6.

[Laplante97] P. A. Laplante, "*Real-time Systems Design and Analysis: An Engineer's Handbook*", 2nd Ed., IEEE Press., 1997.

[Laprie85] J.C. Laprie, "*Dependable Computing and Fault-Tolerance: Concepts and Terminology*", en Proceedings 15d IEEE International Symposium on Fault-Tolerant Computing (FTCS-15), pp. 2-11, Ann Arbor (Michigan, EE.UU.), Junio 1985.

[Laprie92] J.C. Laprie, (Ed.) "*Dependability: Basic concepts ant terminology in English, French, German, Italian and Japanese*". Dependable Computing and Fault Tolerance, Vol. 5, Vienna, Austria, Springer-Verlag, 1992

[Lauterbach02] Laterbach Datentechnik GmbH. "*NEXUS Debugger and Trace for PowerPC*" www.lauterbach.com, 2002

[LIS96] LIS. "*Guide de la süreté de fonctionnement*", Laboratoire d'Ingenierie de la Sureté de fonctionnement (LIS). Cépadúes-Éditions. 1996.

[Madeira94] H. Madeira, M. Rela, F. Moreira, J.G. Silva. "*RIFLE: A General Purpose Pin-level Fault Injector*". In Proc. 1st European Dependable Computing Conference (EDCC-1), pp. 199-216, 1994.

[Madeira00] H. Madeira, D. Costa, and M. Vieira. "*On the emulation of software faults by software fault injection*". In Proceedings of the International Conference on Dependable Systems and Networks, pages 417-426, New York, NY, USA, June 2000. IEEE.

[Madeira03] Madeira, H. et al. "*Emulation of Software Faults: Representativeness and Usefulness*", First Latin-American Symposium on Dependable Computing, LADC, São Paulo, Brasil, October-2003.

[Martínez99] J.R. Martinez, P.J. Gil, G. Martín, C. Pérez, J.J. Serrano. "*Experimental Validation of High Speed Fault Tolerant Systems Using Physical Fault Injection*". In Proc. 7th International Working Conference on Dependable Computing for Critical Applications (DCCA-7), pp. 23 3-249, 1999.

[Miller00] G. Miller, "*The evolution of powertrain microcontrollers and its impact on development processes and tools*". En http://nexus5001.or/rnicrocontrollers_evolution.pdf. Motorola, 2000.

[Moreira03] F. Moreira, R. Maia, D. Costa, N. Duro, P. Rodríguez-Dapena and K. Hjortnaes., "*Static and Dynamic Verification of Critical Software for Space Applications*". Proc. of the Data Systems In Aerospace (DASIA 2003), 25 June 2003. (http://www.eurospace.org/presentations_dasia_2003.htm)

[Motorola99] "*RCPU. RISC Central Processing Unit Reference Manual. Revision 1*". Motorola 1999.

[Motorola00] "*Power is control: MPC555 User's Manual, RISC P0werPCTM*" Motorola, 2000.

[Motorola03] "*MPC5554 Microcontroller Preliminary Product Brief*". Motorola 2003.

[Nexus99] Industry Standards and Technology Organization (IEEE-ISTO). "*The Nexus 5001 Forum™ Standard for a Global Embedded Processor Debug Interface. IEEE-ISTO 5001T11999*" 1999, NY, USA.

[Oberndorf97] Oberndorf, P. (1997). "*COTS and Open Systems. An Overview*". http://www. sei cmu. edu/activities/str/descriptions/cots_body html.

[O'Keefe00] H. O'Keefe, "*IEEE-ISTO 5001™-1999, The Nexus 5001 Forum™*". Standard providing the Gateway to the Embedded Systems of the Future", Ashling Microsystems Limited. Limerick, Irlanda. 2000.

[OSEK05] "*OSEK/VDX. Operating System*". Especificación 2.2.3 disponible en la página web del proyecto OSEK/VDX. http://portal.osek-vdx.org/.

[Pardo01] Juan Pardo. "*Análisis y Benchmarking frente a fallos de aplicaciones del automóvil basadas en Microcontroladores*". Departamento: Informática de sistemas y computadores (DISCA). Grupo de sistemas tolerantes a fallos (GSTF). Programa: Arquitectura y tecnología de los sistemas informáticos. Valencia, Septiembre 2001.

[Pardo04a] Pardo, J., Campelo, J.C, Serrano, J.J. "*Robustness study of an embedded operating system for industrial applications*". The 28th Annual International Computer Software and Applications Software (COMPSAC 2004). ISBN: 0-7695-2209-2 ISSN: 0730-3157. Hong Kong, China. Septiembre, 2004.

[Pardo04b] Pardo, J., Campelo, J.C, Serrano, J.J. "*Reliability study of an embedded operating system for industrial applications*". INFORMATIK 2004. Workshop on Safety, Reliability, and Security of Industrial Computer Systems (WSRS 2004). ISBN:3-88579-379-2 ISSN: 1617-5468. Ulm, Alemania. Septiembre, 2004.

[Pardo05a] Juan Pardo, Juan-Carlos Ruiz, José-Carlos Campelo, Pedro Gil. "*On-chip Debugging-based Fault Emulation for Robustness Evaluation of Embedded Software Components"*. IEEE 11th Pacific Rim International Symposium on Dependable Computing (PRDC 2005). ISBN: 0-7695-2492-3. Changsha, China. Diciembre, 2005.

[Pardo05b] Juan Pardo, José-Carlos Campelo, Pedro Gil. "*Non-Intrusive software fault injection methodology for RTOS robustness testing*". The fifth European Dependable Computing Conference (EDCC 2005). ISBN: 3-540-25723-3 ISSN: 0302-9743. Budapest, Hungría. Abril, 2005.

[Pardo06a] Juan Pardo, José-Carlos Campelo, Juan-Carlos Ruiz, Pedro Gil. "*Temporal Characterization of Embedded Systems Using Nexus"*. The sixth European Dependable Computing Conference (EDCC 2006). Coimbra, Portugal. Octubre2006. ISBN:0-7695-2648-9.

[Pardo06b] Juan Pardo, José-Carlos Campelo, Juan-Carlos Ruiz, Pedro Gil. "*Embedded Software Validation Using On-Chip Debugging Mechanisms*". Software Engineering And Fault Tolerance Book. World Scientific Publishing Co. Pte. Ltd. Series on Software Engineering and Knowledge Engineering.

[Prinetto98] P. Prinetto, M. Rebaudengo, M. Sonza Reorda. "*Exploiting the Background Debugging Mode in a Fault Injection system*". IPDS: The 3rd Annual IEEE International Computer Performance & Dependability Symposium, Durham (NC), September 7-9 1998, p277.

[Rebaudengo99] M. Rebaudengo, M. Sonza Reorda. "*Evaluating the Fault Tolerance Capabilities of Embedded Systems via BDM*". In: Proc. VT599: l7th IEEE VLSI Test Symposium, 1999, pp. 452-457.

[Regan04] P. Regan, S. Hamilton, "*NASA's Mission Reliable*", IEEE
Computer, vol. 37, no 1, pp. 59-68, January 2004.

[Rodríguez99] M. Rodriguez, F. Salles, J.C. Fabre, J. Arlat. "*MAFALDA: Microkernel Assessment by Fault Injection and Design Aid*". In: Proc. European Dependable Computing Conference (EDCC-3), 1999, pp. 143-160.

[Rodríguez02] M. Rodríguez, A. Albinet, J. Arlat, "*MAFALDA-RT: A Tool for Dependability Assessment of Real-Time Systems*", in Proc. IEEE International Conference on Dependable Systems and Networks (DSN 2002), Washington DC (USA), 2002.

[Ruiz04] Ruiz J.-C et a,l "*On Benchmarking the Dependability of Automotive Engine Control Applications*". in Proc. of the 2004 Dependable Systems and Networks, July 2004, Florence (Italy).

[Saiz06] Saiz, L.J. "*Fast and Early Validation of VLSI Systems*". The sixth European Dependable Computing Conference (EDCC 2006). Coimbra, Portugal. Octubre2006. ISBN:0-7695-2648-9.

[Sampson98] J.R. Sampson, W. Moreno, F. Falquez,. "*A technique for Automatic Validation of Fault Tolerant Designs using laser Fault Injection*". In Proc. 28t1 International Symposium on Fault Tolerant Computing (FTCS-28), pp. 162-167, 1998.

[Santos03] L.E. Santos, M.Z. Rela. "*Constraints on the use of Boundary Scan for Fault Injection*". In Proc Latin American Symposium on Dependable Computing, LADC 2003, Sao Paulo, 2003.

[Segall88] 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*". In Proc. 18th Intl. Symposium on fault tolerant computing, pp. 102-107. 1988.

[Sieh97] y. Sieh, O. Tschache, F. Balbach. "*VERIFY: Evaluation of Reliability Using VHDL-Models With Embedded Fault Description*". 27th International Symposium on Fault Tolerant Computing (FTCS-27), pp. 24-27, 1997.

[Siewiorek82] D.P. Siewiorek, RS. Swarz, "*The Theory and Practice of Reliable System Design*", Digital Press, Bedford (Massachusetts, EE.UU.), ISBN 0-93- 237613-4, 1982.

[Smaili04] Idriz Smaili. Monitoring and Debugging of Real-Time Systems: A Survey. Research Report 17/2004, Technische Universitat Wien. Austria. Available online at: http://www.vmars.tuwien.ac.at/php/pserver/extern/docdetail.php?DID=1374&viewmode=paper&year=2004

[Skarin04] D. Skarin, et al. "*Implementation and Usage of the GOOFI MPC565 Nexus Fault Injection Plug-in*" Tech. Report No. 04-08, Chalmers University, Sweden, 2004

[Stence03] R. Stence. "*Nexus 5001 Forum™*". Embedded Systems Conference, San Francisco, CA, EEUU, Abril 2003.

[Steininger02] A. Steininger, C. Scherrer. "*Identifying Efficient Combinations of Error Detecting Mechanisms Based on Results of Fault Injection Experiments*", IEEE Transactions on Computers, vol. 51, no. 2, pp. 235-239, 2002.

[Texas94] Texas Instruments. "*JTAGIMPSD Emulation, Technical Reference*", SPDUO79A, Diciembre 1994.

[Validated02] Validated Software Company. "*MicroC/OS-II MISRA C Compliance Matrix*" Application note AN-2004. http://www.validatedsoftware.com. Weston, Florida, October 23, 2002.

[Vigneron97] C. Vigneron. "*L'injection de fautes. Méthodes et outils existants pour la validation des dispositifs électroniques á vocation sécuritaire*". Internal report INRS ND 2067-169-97, Nancy, France, pp. 609-619, 1997.

[Voas98] Voas, J. G. McGraw. "*Software Fault Injection: Inoculating programs against errors"*. Edit. Wiley. USA, 1998. ISBN: 0-471-18381-4.

[Winter02] Winter M., et al. "*Components for Embedded Software — The PECOS Approach*". In 16th European Conference on Object-Oriented Programming (ECOOP), 2nd. International Workshop on Composition Languages, Málaga, Spain, June 11, 2002.

[Yu0l] Y. Yu. "*A Perspective on the State of Research on Fault Injection Techniques*", Research Report, Mayo 2001.

[Yuste99] P. Yuste, "*Análisis de la aportación de técnicas de tolerancia a fallos en el software sobre la fiabilidad y seguridad de un sistema*". Trabajo de Doctorado, 6 créditos. Departamento de Informática de Sistemas y Computadores, Universidad Politécnica de Valencia. Septiembre 1999.

[Yuste00] P. Yuste, J.C. Campelo, P.J. Gil, J.J. Serrano, "*An approach to dependability modelling of real time systems*". En 25th IFAC Workshop on Real-Time Programming (WRTP 2000), Palma de Mallorca (España) pp. 73-78. Mayo 2000.

[Yuste03a] P.Yuste, L.Lemus, J.J.Serrano, P.J.Gil. "*A Methodology for Software implemented Fault Injection Using NEXUS*". Supplemental volume of the 2003, pp. B12-B13. International Conference on Dependable Systems and Networks, DSN03, 2003. San Francisco, USA.

[Yuste03b] P.Yuste, D. de Andrés, L.Lemus, J.J.Serrano, P.Gil. "*INERTE: Integrated NExus-based Real-Time fault injection tool for Embedded systems*". Proceedings of the 2003 International Conference on Dependable Systems and Networks, DSN03, pp. 669. 2003. San Francisco, USA.

[Yuste03c] P.Yuste, J.C.Ruiz, L.Lemus, P.Gil, "*Non-intrusive software implemented fault injection in embedded systems*" Latin American Symposium on Dependable Computing, LADC2003. 2003. Sao Paulo, Brasil.

[Yuste03d] P. Yuste, D. De Andrés, L. Lemus, J.M. Luján, R. Ors, "*COMODIN: A flexible diesel engine bench control platform*". En Proceedings of 4th Workshop on European Scientific and Industrial Collaboration. Vol. II (WESIC 2003), Miskolc (Hungría), pp. 32 1-328. Mayo 2003.

[Yuste03e] Pedro Yuste. "Contribución a la validación de la Confiabilidad en los sistemas empotrados Tolerantes a fallos". Tesis Doctoral. Departamento de informática de sistemas y Computadores. Universidad Politécnica de Valencia. Dirigida por Dr. Pedro Joaquín Gil Vicente y Dr. Juan José Serrano Martín. Octubre de 2003.