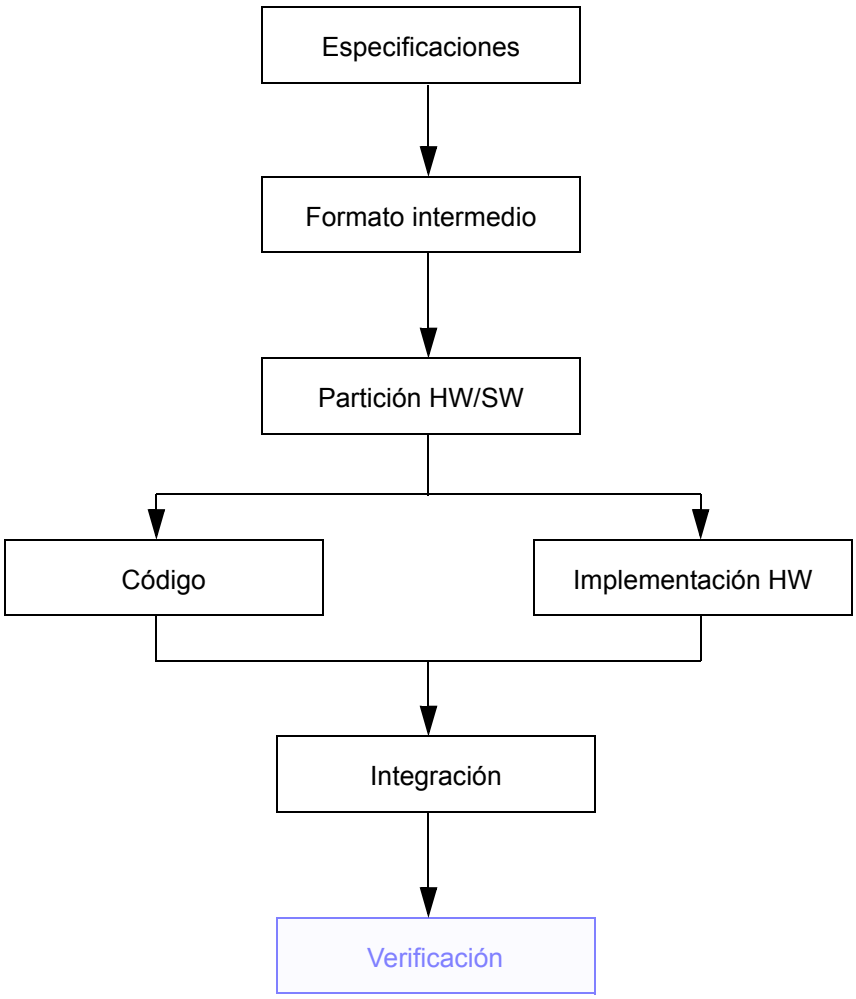
Tema VII

Verificación

Una vez que se ha completado el ciclo de diseño, hay que llevar a cabo un proceso de verificación para asegurar que el sistema obtenido cumple las especificaciones de partida.



7.1. Introducción

A pesar de que ya hemos realizado todos los pasos del flujo de diseño, el proceso de diseño no está completo puesto que cada uno de los pasos está formado realmente por tres tareas bien diferenciadas, como se muestra en la figura 7.1. En primer lugar se realiza la tarea correspondiente al paso del flujo de diseño (generación del modelo, particionado, síntesis hardware, generación de código o integración), para seguidamente realizar una tarea de validación con el fin de asegurar que la tarea en cuestión ha sido llevada a cabo de forma correcta y satisface las restricciones impuestas.

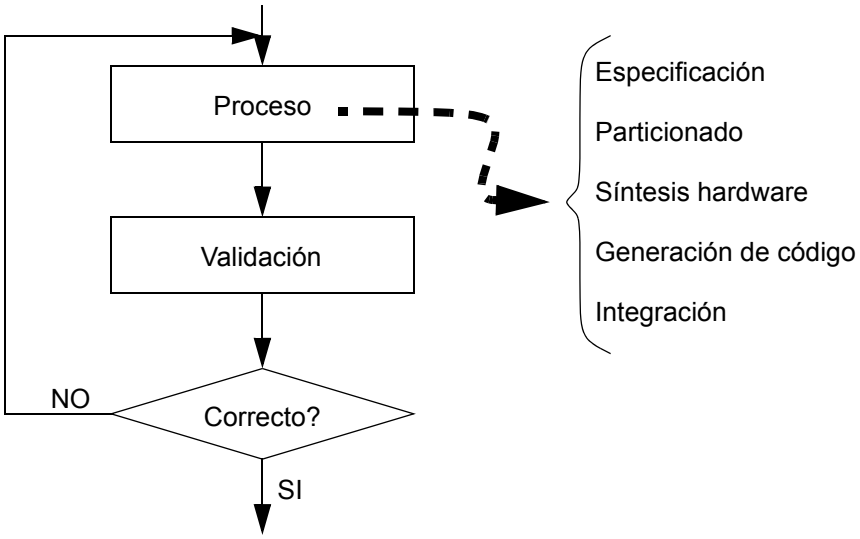


Figura 7.1.- Porción de diseño correspondiente a cada proceso del flujo.

Esta tarea es fundamental ya que evita en la mayor medida de los posible que el prototipo final sea defectuoso, en el sentido de que no cumpla con las especificaciones y/o el comportamiento deseado; y así disminuir los costes de producción. De hecho el 60% de los diseños no funcionan en el primer prototipo, y necesitan uno o más rediseños para afinar el resultado final. Los errores sin detectar pueden aumentar considerablemente los costes de producción e incluso tener consecuencias más extremas. Como ejemplos de estos errores podemos mencionar el bug de los Pentium^{TM} consistente en un fallo de la tabla del multiplicador que no fue verificada de forma completa; la explosión del Ariane 5 que fue debida a una excepción ocurrida durante la conversión de un número flotante de 64 bits en un entero de 16 bits.

Los errores, que son la principal razón de ser de los procesos de validación y verificación, pueden ser agrupados según su fuente, como se puede ver en la figura 7.2. La principal fuente de error se concentra en el proceso de diseño. No obstante, otra fuente de error considerable se encuentran en las especificaciones, ya sea porque sean incorrectas o incompletas (puede darse el caso de que se deseen obtener unas especificaciones incompatibles) o porque durante el proceso de diseño se cambien las especificaciones cuyos cambios no se han considerado en las etapas precedentes del proceso de diseño. Otra fuente de error es debido al mal uso y/o diseño de sistemas IP (subsistemas que serán reutilizables en otros diseños). Por lo tanto,

podemos apreciar que cuanto mayor sea la intervención humana (proceso de diseño, tratamiento de especificaciones y utilización de IPs), mayor es el número de errores que se pueden cometer.

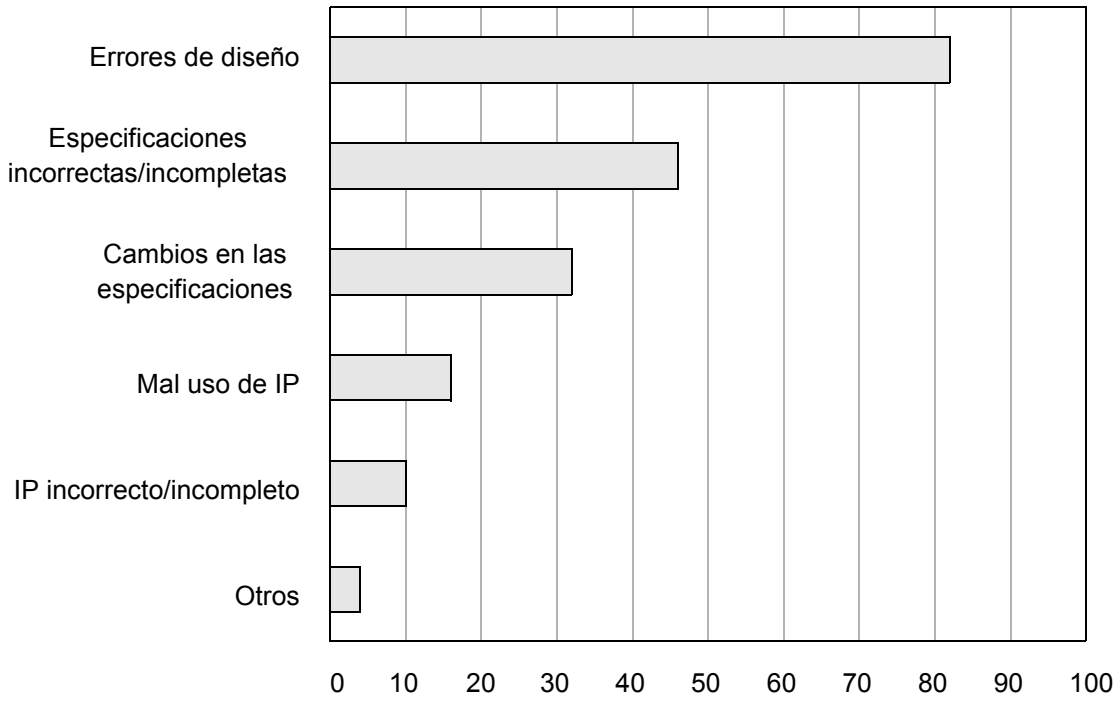


Figura 7.2.- Principales fuentes de errores en el sistema.

A pesar de que se suelen tratar como sinónimos, los términos validación y verificación tienen diferentes matices. Las definiciones que les podríamos dar serían las siguientes:

> La validación es el proceso por el cual se comprueban las propiedades extrínsecas del sistema, es decir, la respuesta a la pregunta ¿estoy haciendo el sistema correcto?
>
> La verificación es el proceso por el cual se comprueban las propiedades intrínsecas del sistema, es decir, la respuesta a la pregunta ¿estoy haciendo el sistema de forma correcta?

Estas comprobaciones han de realizarse en todos los procesos del flujo de diseño, y por tanto en todos los niveles de abstracción. En función de estos niveles, las tareas de verifiación y validación serán realizadas de diferente forma, como se puede ver en la figura 7.3. Como podemos apreciar, cuando el nivel de abstracción es más alto, la verificación se limita a métodos de verificación formal, ya que lo realmente interesante es la funcionalidad del sistema y no sus características. La obtención de las características del sistema implica suposiciones sobre la implementación del sistema, las cuales no son definitivas en estas etapas de diseño; por lo tanto, pueden cambiar, y con ellas las características. A medida que descendemos en la jerarquía, la simulación se vuelve más utilizada como medio de verificación y validación del sistema. Este hecho es debido a que la simulación considera (o puede considerar) elementos que los métodos formales no tienen en cuenta como pueden ser dispositivos parásitos o comportamientos no deseados como son los glitches.

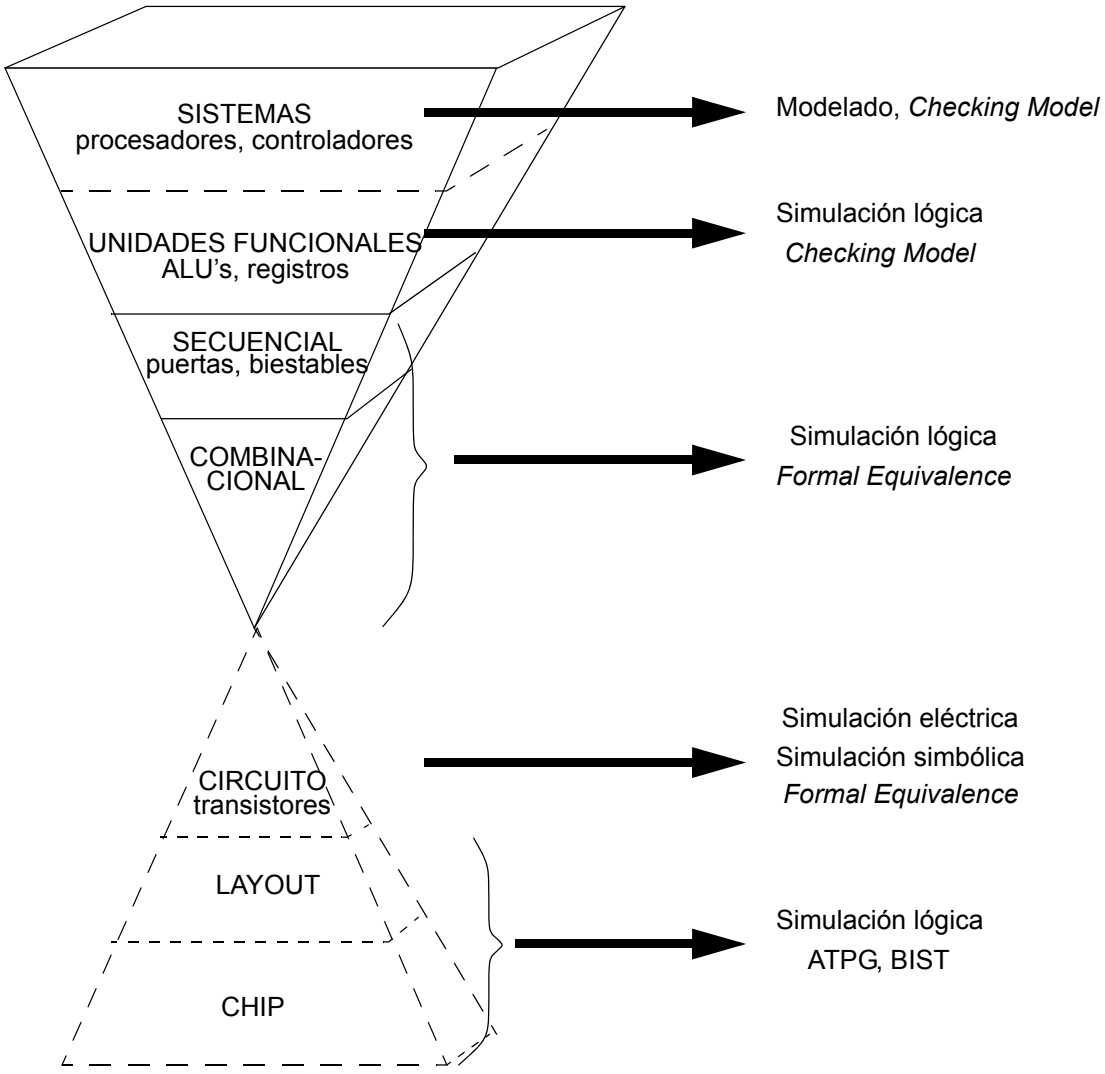


Figura 7.3.- Vista piramidal de los sistemas digitales y sus procesos de verificación más utilizados.

7.2. Métodos de verificación

Ya se han introducido los principales métodos de verificación, pero no todos. Estos métodos los podemos clasificar en tres grandes grupos:

- Métodos formales, en los cuales se trata de determinar analíticamente que las transformaciones realizadas sean correctas.
- Métodos basados en la simulación, en los cuales se trata de determinar la corrección del sistema a través de la simulación, ya sea lógica o eléctrica. Estos métodos suponen la generación de un testbench (banco de pruebas) que deben tener una serie de patrones de test.
- Métodos basados en la emulación, los cuales son similares a la simulación excepto en que la emulación se lleva a cabo sobre prototipos hardware, y la simulación sobre netlist lógicos y/o eléctricos.

Por lo tanto, vamos a tratar en mayor profundidad los métodos correspondientes a los dos primeros grupos.

7.2.1. Métodos formales

La verificación formal utiliza razonamientos rigurosamente matemáticos para demostrar que un sistema mostrará todo o parte del comportamiento especificado. Este hecho requiere la existencia de descripciones formales para las especificaciones e implementaciones.

No obstante, los métodos formales no son una panacea. Aunque se verifique que el netlist de un sistemas satisfaga una especificación formal, no existe una completa seguridad de que el sistema físico se comporte como se pretende. Este hecho suele ser debido a que estamos tratando con modelos abstractos (independientemente del nivel en el que nos estemos moviendo). Luego, como mínimo es necesario realizar una validación experimental para tener la seguridad anterior.

7.2.1.1. Especificaciones formales.

Para poder utilizar unos razonamientos matemáticos, se deben especificar formalmente tanto las propiedades que se desean verificar como el sistema a verificar. En este caso, el sistema va a ser modelado con un diagrama de estados, como vimos en el tema II.

En cambio, las propiedades van a ser especificadas utilizando una lógica temporal (ya que las propiedades suelen tener una connotación temporal), la cual v a ser tratada a continuación.

- Lógica temporal

La lógica temporal está dirigida a verificar propiedades que cambian con el tiempo; por lo tanto, estamos ante una lógica (usualmente proposicional) enriquecida con operadores temporales que permite razonar sobre cómo cambiarán las afirmaciones en el tiempo. Dentro de las lógicas temporales, vamos a centrarnos en la lógica de árboles de computación, CTL (Computation Tree Logic), que utiliza un modelo discreto de tiempos en el que el tiempo se puede dividir en más de un posible futuro partiendo desde un mismo estado inicial, como podemos ver en la figura 7.4. En este caso, y considerando la porción de diagrama de estado, tenemos un ciclo entre los estados p1 y p3; otro camino es el que continua con el estado p2 y después los estados p4 o p5.

El sistema viene determinado por el vector M=(S, R, L), donde S es el conjunto de estados, R es la relación de próximo estado y L es el conjunto de proposiciones que son ciertas en un estado dado (en forma de etiquetas). Sólo existe la restricción de que el sistema esté completamente especificado, es decir, que la relación de proximo estado sea total; este hecho es debido a que las fórmulas CTL no tienen interpretación para estados sin sucesores (próximos estados). Las fórmulas CTL se definen recursivamente como (el orden es importante):

- Cualquier proposición es una fórumla CTL
- Si f1 y f2 son fórmulas CTL, también lo son:

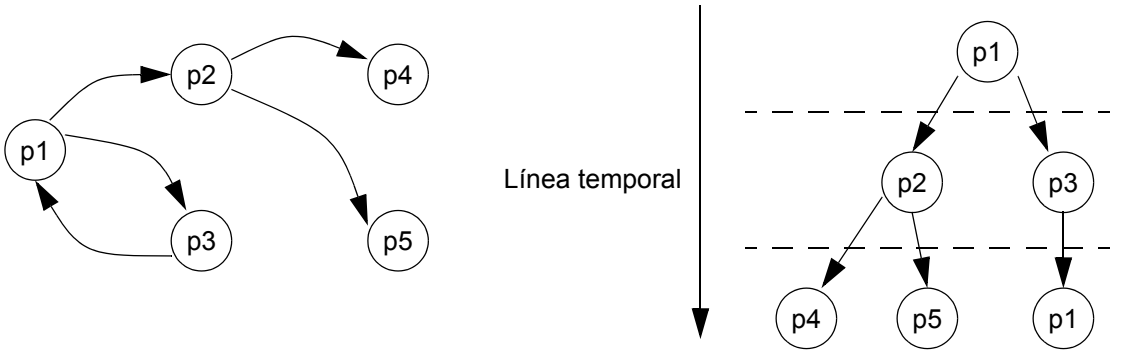


Figura 7.4.- Representación gráfica de un CTL.

¬f1 --> negación de f1
f1∧ f2 --> operación Y
f1 ∨ f2 --> operación O
f1 => f2 --> Después de satisfacerse f1, se satisfará f2
AX f1 --> En todos los próximos estados se mantiene f1
EX f1 --> En algún próximo estado se mantiene f1
A [f1 ∪ f2] --> Para todos los caminos se mantiene f1 hasta que se mantiene f2
E [f1 ∪ f2] --> Para algún camino se mantiene f1 hasta que se mantiene f2

- Otros operadores útiles, basados en los anteriores son los siguientes:
  AF f = A [true ∪ f] --> En todos los caminos se mantiene f
  EF f = E [true ∪ f] --> En algún camino se mantiene f
  AG f = ¬EF¬f --> En todos los caminos siempre se mantiene f
  EG f = ¬AF¬f --> Hay algún camino que siempre se mantiene f

Si consideramos una fórmula CTL f, una estructura M=(S, R, L) y un estado $s_0 \in S$, la relación |= define una verdad formal y es definida recursivamente de la siguiente forma (de nuevo el orden es importante):

- $M,s_0 \models f \Leftrightarrow f \in L(s_0)$
- $M,s_0 \models \neg f \Leftrightarrow f \notin L(s_0)$
- $M,s_0 \models f1 \wedge f2 \Leftrightarrow M,s_0 \models f1$ y $M,s_0 \models f2$
- $M,s_0 \models AX\ f \Leftrightarrow$ para todo $t \in S \mid (s_0, t) \in R \Rightarrow M,t \models f$
- $M,s_0 \models EX\ f \Leftrightarrow$ existe $t \in S \mid (s_0, t) \in R \Rightarrow M,t \models f$
- $M,s_0 \models A\ [f1 \cup f2] \Leftrightarrow$ existe $i \geq 0 \mid M,s_i \models f2 \wedge$ para todo j= 0, ..., i-1 $M,s_j \models f1$ en todos los caminos
- $M,s_0 \models E\ [f1 \cup f2] \Leftrightarrow$ existe $i \geq 0 \mid M,s_i \models f2 \wedge$ para todo j= 0, ..., i-1 $M,s_j \models f1$ en algún camino

En la figura 7.5 podemos apreciar diferentes árboles lógicos de computación junto con la fórmula que se cumple en dicho árbol. Así podemos apreciar que los árboles no tienen porqué ser homogéneos, es decir, cada estado podrá tener diferente número de transiciones. Dicha figura nos sirve para indicar las principales diferencias entre las letras de los operadores:

- A -> implica todos los caminos desde el estado considerado
- E -> implica algún camino desde el estado considerado
- X -> implica un siguiente estado (según la letra inicial A o E) desde el estado considerado
- F -> implica algún estado de un camino
- G -> implica todos los estados de un camino

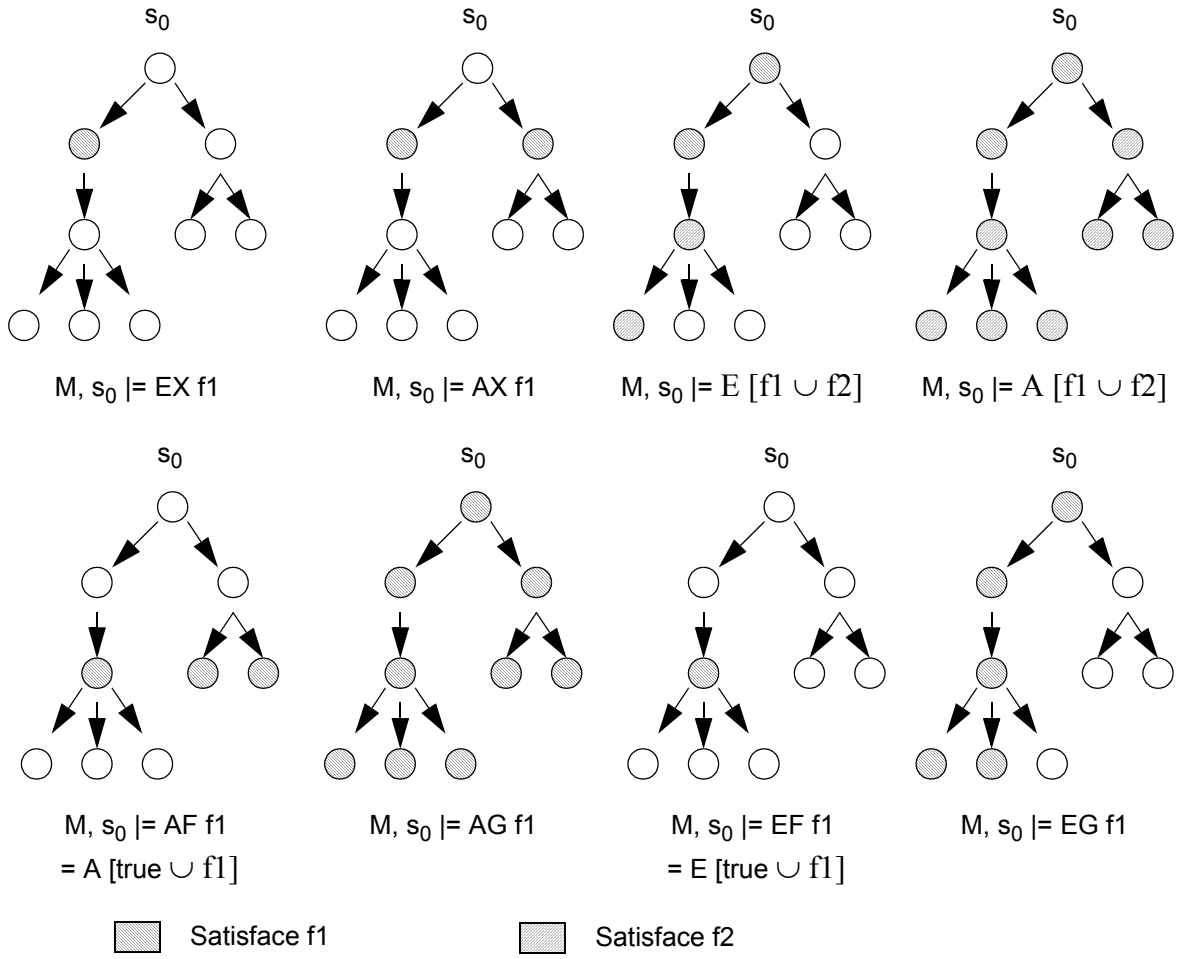


Figura 7.5.- Ejemplos de fórmulas lógicas temporal en función del árbol lógico temporal.

7.2.1.2. Checking Model (Chequeo de modelo)

La meta del chequeo de modelo consiste en determinar las propiedades que cumple un sistema determinado. Para ello se deben realizar una serie de tareas. Dichas tareas, por el orden de realización son las siguientes:

- Modelado y especificación del sistema. Con esta tarea debemos obtener un modelo del sistema que estemos tratando. Dentro de las diferentes formas de modelado que estudiamos en el tema 2, este proceso está dirigido hacia los diagramas de estados. Para solventar el problema de la explosión del número de estados cuando la compleji-

dad del sistema aumenta, se usarán métodos símbolicos o BDDs (Binary Decission Diagram).

- Especificación de las propiedades. Con esta tarea debemos especificar las propiedades que deseemos verificar en un lenguaje formal. Este lenguaje formal suele ser la lógica temporal.
- Verificación propiamente dicha. Con esta tarea vamos a identificar los estados que cumplen las propiedades especificadas.

Como los primeros dos puntos ya han sido tratados, vamos a centrarnos en el algoritmo más utilizado para abordar la tercera tarea. Esta tarea se puede enunicar como

> dados una estructura M={S, R, L} y una fórmula lógica temporal f, determinar el conjunto de estados que satisfacen f ({s $\in$ S : M,s |= f})

Para lograr este objetivo, en primer lugar se debe dividir la fórmula en las diferentes subfórmulas de jerarquía inferior. Se obtiene la lista de todas las subfórmulas de tal modo que se empieza por las subfórmulas más símples, y se acaban con las más complejas, es decir, con la fórmula. Como último paso, se etiquetan los estados que cumplen cada una de las subfórmulas anteriores. Por último, el conjunto que satisface la fórmula serán todos los estados que contengan la última etiqueta.

Veamos el ejemplo de un controlador de un semáforo de una intersección de cuatro calles como la mostrada en la figura 7.6. En este caso los semáforos de direcciones opuestas encenderán las mismas luces, por lo que tendremos únicamente dos salidas. El conjunto de estados estará formado por los pares de luces de las dos direcciones (ns, eo).

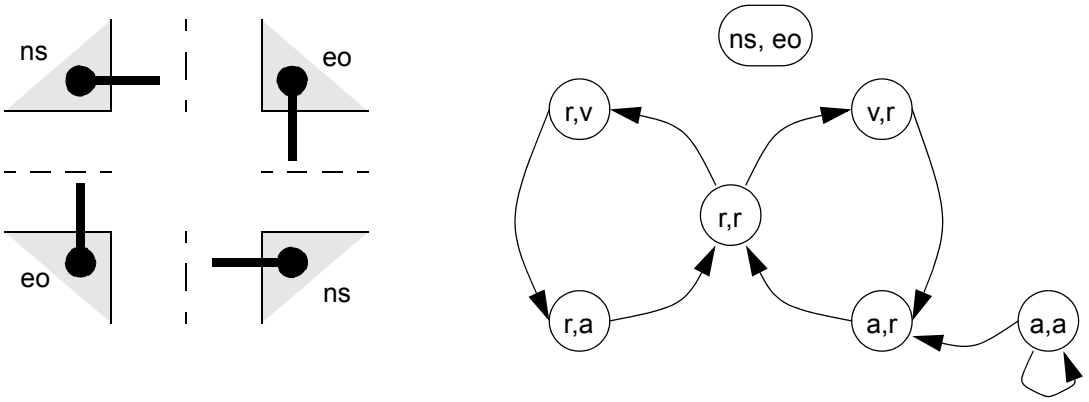


Figura 7.6.- Esquema de la intersección modelada y su modelo en diagrama de estados.

Una propiedad importante que debería cumplir nuestro controlador será que siempre exista una luz roja, en al menos una de las direcciones. Dicha propiedad la podríamos enunciar de la siguiente forma:

$$P = AG ([ns = r] \vee [eo = r])$$

Tenemos que notar que M $\not\models$ P, ya que el estado (a, a) no lo cumple, luego M, (a, a) $\not\models$ P. No obstante, uno puede apreciar que todas las transiciones de estado reales (que implican un cambio real de estado y no un autolazo) cumplen dicha propiedad. Una formulación equivalente de dicha propiedad es

$$M \models ([ns = r] \wedge [eo = r]) \Rightarrow AG\ ([ns = r] \vee [eo = r])$$

Esta fórmula se puede identificar como la formulación genérica de la propiedad de *safety* (nada malo debe suceder), de la forma

$$M \models P_{inicial} \Rightarrow AG\ P_s$$

donde $P_{inicial}$ y P_s deben ser propiedades que no contienen operadores temporales. De tal modo que partiendo de todos los estados que satisfagan $P_{inicial}$, en todos los estados sucesores y para todos los posibles caminos se cumplirá P_s.

Para chequear la propiedad anterior en el controaldor del semáforo, en primer lugar se debe dividir la fórmula en varias subfórmulas por orden de jerarquía. Así tendremos las siguientes fórmulas:

- [ns = r] = p1
- [eo = r] = p2
- [ns = r] ^ [eo = r] = p3
- [ns = r] ∨ [eo = r] = p4
- AG ([ns = r] ∨ [eo = r]) = p5
- ([ns = r] ^ [eo = r]) => AG ([ns = r] ∨ [eo = r]) = p6

Por lo tanto, dicha propiedad será cumplida por todos los estados del controlador de una operación normal, ya que el estado en el que las dos direcciones tienen el color ambar. Esta situación se muestra en la figura 7.7.

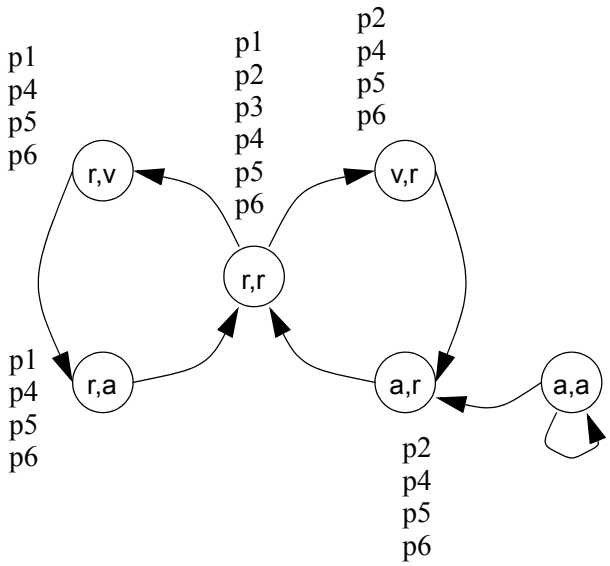


Figura 7.7.- Modelo en diagrama de estados, junto con las etiquetas de las subfórmulas.

No obstante, podemos decir que el sistema cumple la propiedad anterior ya que el estado que no la cumple nunca es alcanzado desde la proposición inicial ([ns = r] ^ [eo = r]).

Veamos un segundo ejemplo de utilización del chequeo de modelo. Otra propiedad importante es la siguiente:

$$EF ([ns = v] \vee [eo = v])$$

La cual implica que existirá un camino en el que se llegará a que una de las direcciones estará habilitada. La lista de subfórmulas es la siguiente:

- [ns = v] = q1
- [eo = v] = q2
- [ns = v] ∨ [eo = v] = q3
- EF ([ns = v] ∨ [eo = v]) = q4

Luego, en la figura 7.8 mostrados el diagrama de estados con las etiquetas de las diferentes subfórmulas. En dicha figura podemos apreciar que el sistema modelado cumple con dicha propiedad, ya que todos sus estados la cumplen.

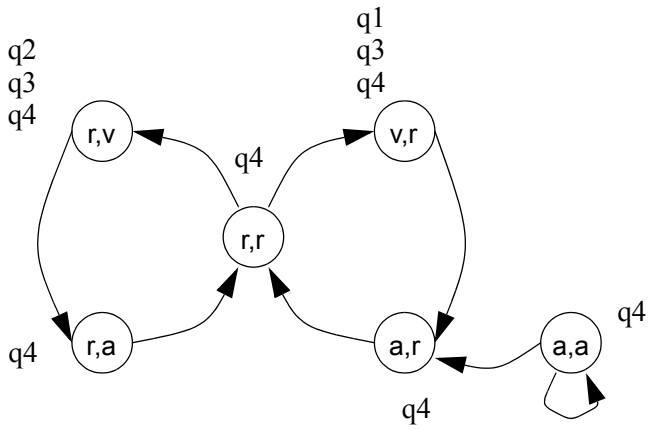


Figura 7.8.- Modelo en diagrama de estados, junto con las etiquetas de las subfórmulas.

7.2.2. Métodos basados en la simulación y emulación

La verificación formal, que hemos tratado en el apartado anterior, suelen estar dirigida a niveles superiores de la jerarquía (como se observaba en la figura 7.3), estando todos ellos por encima del nivel de transferencia de registros (RTL). Para el resto de niveles de abstracción, el método más utilizado para la verificación es la simulación, tanto eléctrica como lógica.

Para realizar una simulación, debemos tener una descripción estructural del sistema. Así que con la simulación debemos verificar dos hechos fundamentalmente:

- la comunicación entre los diferentes bloques del sistema,
- y la funcionalidad de cada uno de dichos bloques.

Para llevar a cabo la simulación es necesario disponer de tres objetos:

- un banco de patrones de test (o *testbench*),
- un modelo para el sistema que se desea simular, y

- un modelo de referencia para comprobar los resultados.

como se muestra en la figura 7.9.

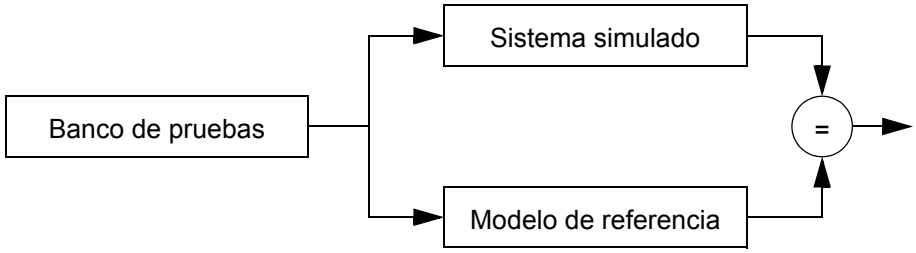


Figura 7.9.- Esquema de un entorno de simulación

El banco de pruebas no es más que una secuencia en los puertos de entrada al diseño bajo test (DUT, Design Under Test). El comportamiento del DUT con dichos valores en los puertos de entrada es comparado con un modelo de referencia para determinar el grado de corrección del DUT.

Uno de los principales problemas en la simulación consiste en obtener unos patrones de test adecuados para la verificación del comportamiento bajo test. De hecho, los principales comportamientos que se suelen simular (y para los que se suelen utilizar patrones de test diferentes) son:

- Transacciones; en las que se comprueban buses y circuitos de interfaz, es decir, las comunicaciones del sistema.
- Instrucciones; en el caso de tener una arquitectura ISA (Instruction Sets Architecture), en la que hay que garantizar que las diferentes instrucciones cumplen la misión encomendada.
- Errores atípicos; en los que se comprueban secuencias cuya aparición no es típica en el funcionamiento del sistema, como podría ser la consecución de dos incios de operación sin que haya acabado la primera de ellas. Para ello, el patron de test suele ser aleatorio.
- Comportamiento funcional; debido a que el funcionamiento de un sistema suele ser grande, esta verificación suele estar limitada.

Dos características claves en lo que respecta a la verificación son la controlabilidad y la observabilidad. La controlabilidad implica que la operación del sistema es controlable a través de señales específicas de control de test; mientras que la observabilidad implica que la operación es observable a través de puntos de monitorización en el sistema. En función de la observabilidad, podemos distinguir tres estrategias de simulación:

- Caja negra (*blackbox*), en la cual la observabilidad se limita a las señales de salida, por lo cual el diseño es totalmente invisible al verificador.
- Caja blanca (*whitebox*), en la cual la observabilidad es total ya que el verificador tiene acceso a todos los puntos del sistema.
- Caja gris (*graybox*), en la cual la observabilidad se encuentra entre los dos casos ante-

riores, de tal forma que la observabilidad se realiza a través de las salidas del sistema (como en el caso de la caja negra) y los puntos más críticos (en lugar de todos los puntos del sistema como en el caso de la caja blanca).

El principal problema de la estrategia de caja negra es el elevado tiempo de debugging ya que determinar la parte del sistema que no supera el proceso de verificación es muy complicado si únicamente podemos observar los puertos. En cambio, el principal problema de la caja blanca radica en el elevado volumen de información disponible debido a la observabilidad de todos los puntos del sistema. Luego la estrategia más adecuada suele ser la caja gris ya que aumenta la observabilidad de los estrictamente puertos de salida pero dicho aumento no es lo suficiente elevado como para resultar contraproducente.

7.3. Tolerancia a fallos

Algunos sistemas se requiere (o desea) que sean tolerantes a fallos, ¿pero que se entiende por un sistema tolerante a fallos?

> Un sistema tolerante a fallo es aquel que muestra un comportamiento de fallo bien definido cuando sus componentes fallan, o enmascaran los fallos de los componentes al usuario (es decir, continua mostrando el comportamiento estándar especificado a pesar del fallo de algún componente)

El número de usuarios que demanda este tipo de sistemas está aumentando progresivamente. Algunos ejemplos de los sistemas que deberían tener esta capacidad pueden ser los siguientes: sistemas de procesado en las transacciones on-line, sistemas de control de procesos o sistemas de comunicaciones basadas en computadores.

7.3.1. Conceptos básicos

De forma previa al tratamiento de esta disciplina, diseño de sistemas tolerantes a fallos, debemos clarificar una serie de definiciones.

En primer lugar, tenemos tres definiciones relacionadas con los componentes, las cuales son servicios, servidores y dependencia:

> Un servicio especifica una colección de operaciones cuya ejecución puede ser disparada por los usuarios del servicio o por el paso del tiempo

La ejecución del servicio puede resultar en salidas a los usuarios o en el cambio del estado del servicio.

> Un servidor es el componente que implementa un servicio sin exponer a los usuarios la representación del estado interno del servicio y los detalles de su implementación

Por lo que un servidor será visto por un usuario como una caja negra, ya sea hardware o software.

> Un servidor u depende de otro r si el comportamiento correcto de u depende de que el comportamiento de r sea correcto.

En este caso, el servidor u es llamado usuario (o cliente) y r es llamado recurso. Esta relación suele ser representada por un grafo acíclico en el que los nodos representan servidores y las flechas representan relaciones. El usuario se dice que está en un nivel de abstracción superior que el recurso, por lo que suele estar representado encima del recurso (o lo que es lo mismo, el grafo irá del nivel superior al inferior de abstracción). Supongamos que se desea implementar un servidor de ficheros (denominado f), el cual dependerá de un servidor de entrada/salida (denominado s) y un servidor de colocación (denominado d); el grafo de dependencia del servidor f será el mostrado en la figura 7.10.

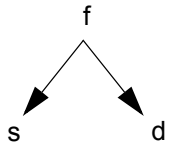


Figura 7.10.- Grafo de relación de dependencia.

Si estamos hablando de comportamiento correcto, debemos identificar que es lo que se entenderá por *correcto*.

> Un servidor diseñado para proveer un determinado servicio es correcto si, en respuesta a sus entradas, se comporta de manera consistente con las especificaciones del servicio.
>
> En caso contrario, el servidor, o el servicio del servidor, no será correcta, o habrá un fallo del servidor, si no se comporta de la manera especificada.

No obstante, se pueden dar muchas formas de no comportarse de manera especificada, de tal forma que podemos distinguir diferentes tipos de fallos:

> Un fallo de omisión es aquel en el que el servidor omite responder a una entrada.
>
> Un fallo de temporización se produce cuando la respuesta del servidor es funcionalmente correcta pero se produce fuera de tiempo, es decir, fuera del intervalo temporal especificado. Así podemos encontrar fallos de temporización temprano o tarde.
>
> Un fallo de respuesta se produce cuando el servidor responde de forma incorrecta, ya sea por valor (fallo de valor) o por la transición de estado (fallo de transición de estado)
>
> Un fallo de *crash* se produce cuando el servidor muestra un fallo de omisión a todas las entradas, desde el primer fallo de omisión, hasta que se reinicia. Hay varios fallos de *crash*: amnesia, cuando el estado después de la reinicialización no depende de la situación de las entradas antes del fallo; amnesia parcial, cuando al reiniciar, parte del estado es el mismo que el que tenía antes del fallo; pausa, cuando reempieza en el estado que tenía antes del fallo; *halting*, cuando el sistema nunca reempieza.

Como ejemplos de fallos de omisión, podemos encontrar los fallos producidos en algún servicio de comunicación, los cuales son transparentes para el usuario (si se realiza un ping a una máquina, los paquetes enviados son divididos en: enviados, recibidos y perdidos; los paquetes

perdidos, en el caso de que los hubiese, serían los fallos de omisión). Un fallo de temporización podría ser un tiempo excesivo para copiar datos desde un CD al disco duro debido a un defecto físico del CD. Un fallo de amnesia parcial o de pausa podría ser cuando un procesador de texto se *cuelga*, al ejecutarse de nuevo, tendremos una versión recuperada del documento que se estaba editando, que puede coincidir con las últimas modificaciones antes del fallo (fallo de pausa) o una versión anterior (fallo de amnesia parcial). En cambio, cuando el sistema operativo suele colgarse, el fallo suele ser de amnesia ya que hay que reiniciar el sistema completo.

Una de las posibles definiciones de sistema tolerante a fallos es la de conocer el comportamiento de fallo del sistema; por lo tanto, esta situación es importante. De hecho, en un sistema tolerante a fallos hay que indicar las especificaciones estándar del sistema (cuando la operación es correcta) y las especificaciones de fallos (cuando se produce algún fallo en la operación, denominada ésta última como semántica de fallos).

7.3.2. Propagación jerárquica

Hasta el momento, hemos considerado los posibles fallos de un sistema (o un bloque) aislado, y así se debe conocer el comportamiento del fallo y/o enmascarar dicho comportamiento. No obstante, la arquitectura de cualquier sistema empotrado presupone un diseño jerárquico, por lo tanto hay que determinar cómo se propagan los fallos a través de la jerarquía del sistema global. De hecho, un fallo de un tipo se puede transformar en un fallo de diferente tipo en un nivel superior de la jerarquía. Por ejemplo, supongamos que tenemos una línea de transmisión con dos servidores diferentes: la transmisión física y el chequeo de datos, tal como se muestra en la figura 7.11. Si el chequeo es tal que no hace nada cuando los datos que llegan son erróneos; un fallo de valor en la transmisión física se convertirá en un fallo de omisión en el servidor del chequeo de datos.

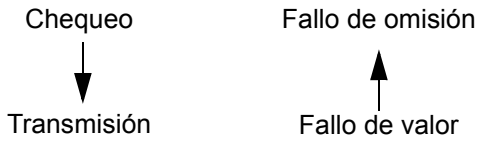


Figura 7.11.- Ejemplo de la línea de trnasmisión

En general, si un servidor depende de un recurso con una determinada semántica de fallo, entonces el servidor tendrá una semántica arbitraria, a menos que dicho servidor tenga algún medio de chequear que la respuesta del recurso sea correcta o no. No obstante, diseñar un chequeador de errores es una tarea compleja que usualmente no compensa, por lo que usualmente se tiende a utilizar una semántica de fallos fuerte, como fallos *crash*, de omisión o *performance*.