



Tema 13. Fiabilidad y tolerancia de fallos

1. Introducción y objetivos
2. Prevención y tolerancia de fallos
3. Redundancia estática y dinámica
 - a. Programación con N versiones
 - b. Bloques de recuperación
 - c. Excepciones
4. Seguridad, fiabilidad y confiabilidad



Introducción y objetivos



- El aspecto que queremos tratar es el de la extremada fiabilidad exigida a los sistemas de tiempo real
- Veremos cuáles son los factores que afectan a la fiabilidad de un sistema
- También veremos algunas técnicas para tolerar fallos en el software



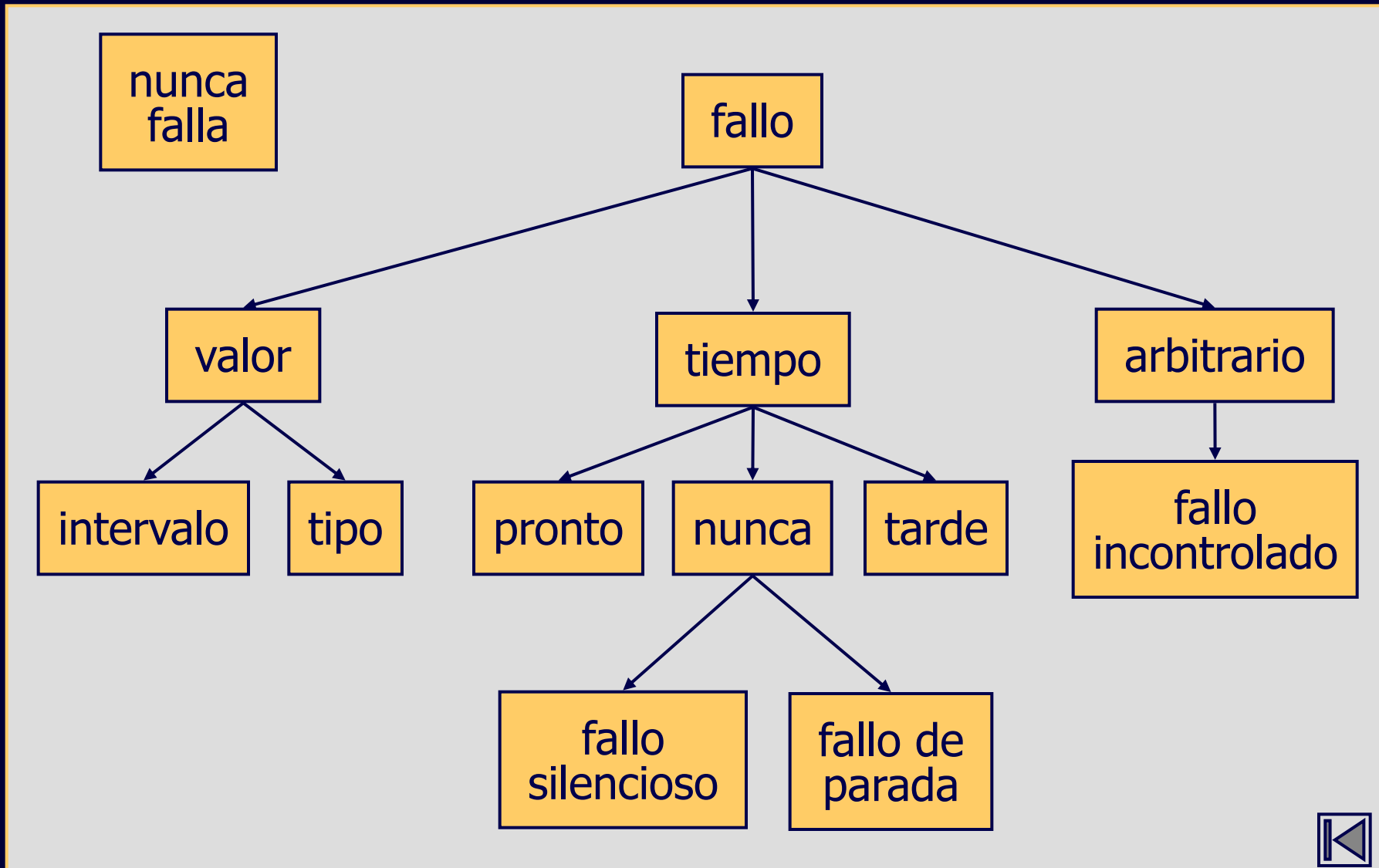
- Los fallos de funcionamiento de un sistema pueden tener su origen en:
 - Una especificación inadecuada
 - Diseño defectuoso del software y/o el hardware
 - Averías en el hardware
 - Interferencias en las comunicaciones
 - transitorias o permanentes
- Nos centraremos en el estudio de los errores en el software



- La **fiabilidad** (*reliability*) de un sistema es una medida de su conformidad con una especificación autorizada de su comportamiento
- Una **avería** (*failure*) es una desviación del comportamiento de un sistema respecto a su especificación
- Las averías se manifiestan en el comportamiento externo del sistema, pero son el resultado de **errores** (*errors*) internos
- Las causas mecánicas o algorítmicas de los errores se llaman **fallos** (*faults*)
- Los fallos pueden ser consecuencia de averías en los componentes del sistema



- Fallos **transitorios**:
 - desaparecen por sí solos al cabo de un tiempo
 - ejemplo: interferencias en las comunicaciones
- Fallos **permanentes**:
 - duran hasta que se reparan
 - ej.: un cable roto, un defecto en el software
- Fallos **intermitentes**:
 - fallos transitorios que, además, ocurren de vez en cuando
 - ejemplo: calentamiento de un componente del hardware





Prevención y tolerancia de fallos



- Hay dos formas de aumentar la fiabilidad de un sistema:
 - **Prevención** de fallos
 - se trata de evitar, antes de que el sistema entre en funcionamiento, que se introduzcan fallos
 - **Tolerancia** de fallos
 - se trata de conseguir que el sistema continúe funcionando aunque se produzcan fallos
- En ambos casos el objetivo es desarrollar sistemas con modos de fallo bien definidos
- No son incompatibles



- Se realiza en dos etapas:
 - **Evitación** de fallos
 - se trata de impedir que se introduzcan fallos durante la construcción del sistema
 - **Eliminación** de fallos
 - consiste en encontrar y corregir los fallos que se producen en el sistema una vez construido



- Hardware:
 - Utilización de componentes fiables
 - Técnicas rigurosas de montaje de subsistemas
 - *Apantallamiento* del hardware
- Software:
 - Especificación rigurosa (formal) de requisitos
 - Métodos de diseño comprobados
 - Lenguajes con abstracción de datos y modularidad
 - Utilización de entornos de desarrollo con computador (*CASE*) adecuados para gestionar los componentes



- **Comprobaciones:**
 - Revisiones del diseño
 - Verificación de los programas
 - Inspección del código
- **Pruebas (*tests*):**
 - Son necesarias, pero insuficientes:
 - nunca llegan a ser exhaustivas
 - sólo sirven para mostrar que hay errores, pero no que **no** los hay
 - muchas veces es imposible reproducir las condiciones reales
 - los errores de especificación no se detectan



- Los componentes del hardware fallan, a pesar de las técnicas de prevención
- La prevención es insuficiente si:
 - la frecuencia o la duración de las reparaciones es inaceptable
 - no se puede detener el sistema para efectuar operaciones de mantenimiento
- La alternativa es utilizar las técnicas de **tolerancia de fallos**



- Formas:
 - **Tolerancia completa** (*fail operational*)
 - El sistema sigue funcionando, al menos durante un tiempo, sin perder funcionalidad ni prestaciones
 - **Degradación elegante** (*fail soft*)
 - El sistema sigue funcionando con una pérdida parcial de funcionalidad o prestaciones hasta la reparación del fallo
 - **Parada segura** (*fail safe*)
 - El sistema se detiene en un estado que asegura la integridad del entorno hasta que se repare el fallo
- El grado y la forma de tolerancia de fallos dependen de la aplicación



Redundancia estática y dinámica



- La tolerancia de fallos se basa en la **redundancia**
- Se utilizan componentes adicionales para:
 - **detectar** los fallos y
 - **recuperar** el funcionamiento correcto
- Esto aumenta la complejidad del sistema
 - y puede introducir fallos adicionales
- Es mejor separar del resto del sistema a los componentes tolerantes de fallos



- Redundancia **estática**
 - los componentes redundantes están siempre activos
 - se utilizan para enmascarar los fallos
 - Redundancia Modular N (triple o más)
- Redundancia **dinámica**
 - los componentes redundantes se activan cuando se detecta un fallo
 - se basa en la **detección** y posterior **recuperación** de los fallos



- Técnicas para detectar y corregir errores de diseño
 - Redundancia **estática**
 - Programación con N versiones
 - Redundancia **dinámica**
 - Dos etapas: detección y recuperación de fallos
 - **Bloques de recuperación**
 - proporcionan recuperación *hacia atrás*
 - **Excepciones**
 - proporcionan recuperación *hacia adelante*

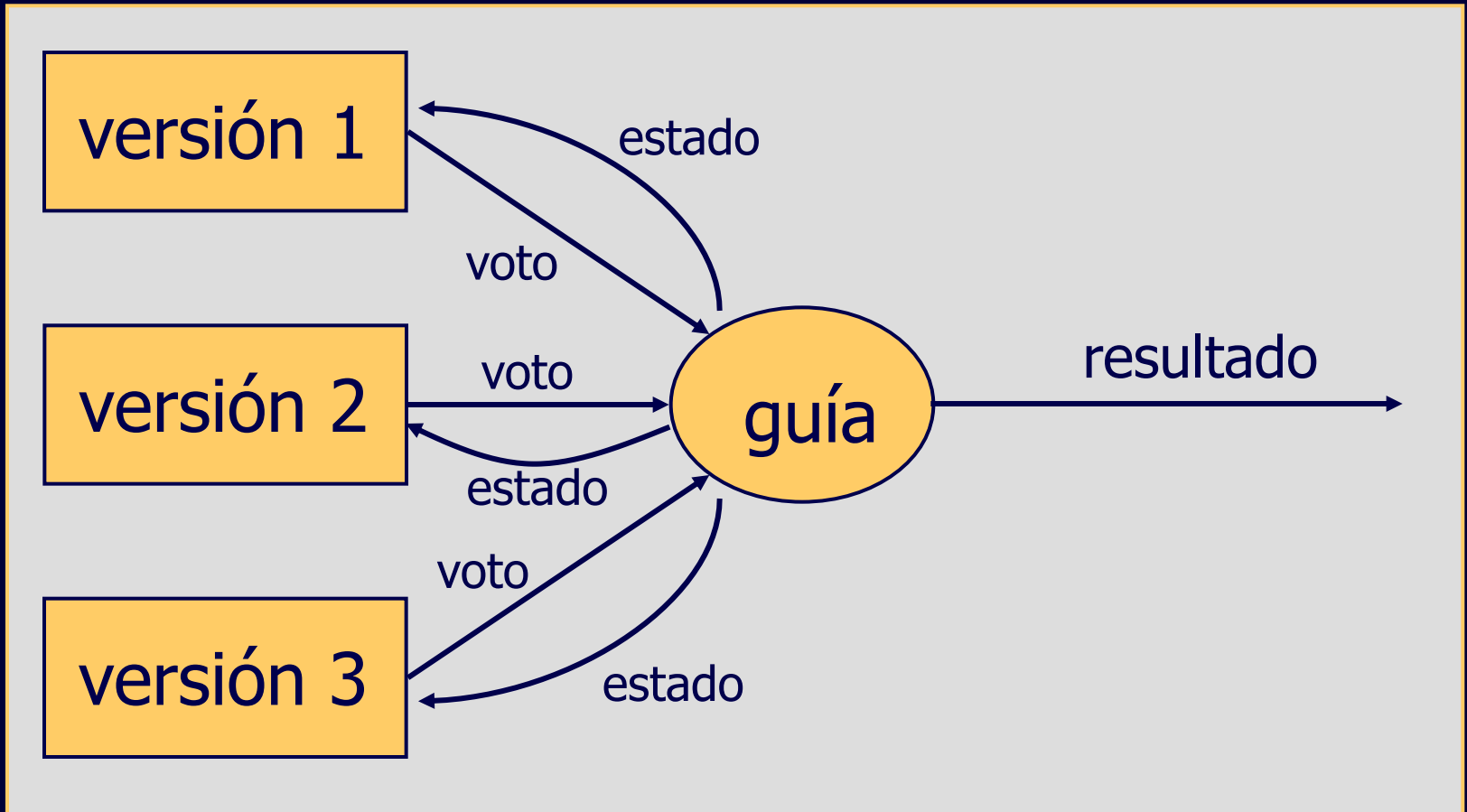




Programación con N versiones

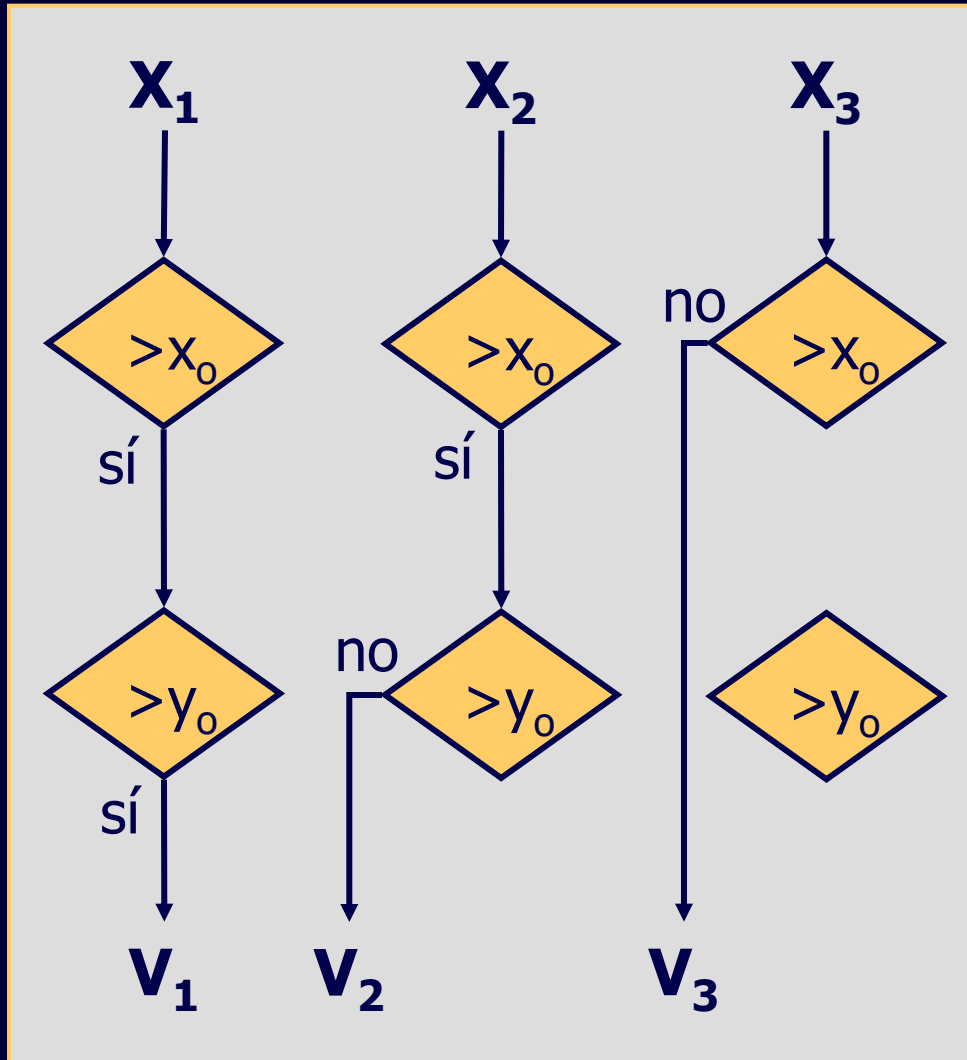


Programación con N versiones





Comparación consistente



- La comparación de valores reales no es exacta
- Cada versión produce un resultado correcto, pero diferente de las otras
- No se arregla permitiendo una tolerancia $\pm\Delta$



Problemas de la programación con N versiones

- La correcta aplicación de este método depende de:
 - **Especificación inicial**
 - si hay un error en la especificación, aparecerá en todas las versiones
 - **Desarrollo independiente**
 - no debe haber interacción entre los equipos
 - no es seguro que distintos programadores cometan errores independientes
 - **Presupuesto suficiente**
 - los costes de desarrollo se multiplican
 - el mantenimiento es también más costoso
- Se ha utilizado en sistemas de aviónica críticos



Redundancia dinámica en software



- Los componentes redundantes sólo se ejecutan cuando se detecta un error
- Se distinguen cuatro etapas:
 1. **Detección de errores**
 2. **Evaluación y confinamiento de los daños**
 3. **Recuperación de errores**
 - se trata de llevar el sistema a un estado correcto, desde el que pueda seguir funcionando
 4. **Reparación de fallos**
 - aunque el sistema funcione, el fallo puede persistir y hay que repararlo



- Por parte del **entorno de ejecución**
 - hardware (p.ej. Instrucción ilegal)
 - núcleo o sistema operativo (p.ej. puntero nulo)
- Por parte del software de aplicación
 - duplicación (redundancia con 2 versiones)
 - comprobaciones de tiempo (vigilante)
 - funciones inversas
 - códigos detectores de error (sumas)
 - validación de estado *razonable*: estática (rango) o dinámica (salidas consecutivas)
 - validación estructural (integridad de listas)



- El objetivo es **confinar** los daños causados por un fallo a una parte acotada del sistema
- Se trata de estructurar el sistema de forma que se minimice el daño causado por los componentes defectuosos
 - compartimentos estancos, cortafuegos
- Técnicas
 - **Descomposición modular**: confinamiento estático
 - **Acciones atómicas**: confinamiento dinámico



- Es la etapa más importante
- Se trata de situar al sistema en un estado correcto desde el que pueda seguir funcionando
- Hay dos maneras de proceder:
 - recuperación **hacia adelante**
 - se avanza desde un estado erróneo haciendo correcciones sobre partes del estado
 - recuperación **hacia atrás**
 - se retrocede a un estado anterior correcto que se ha guardado previamente



Recuperación de errores hacia adelante

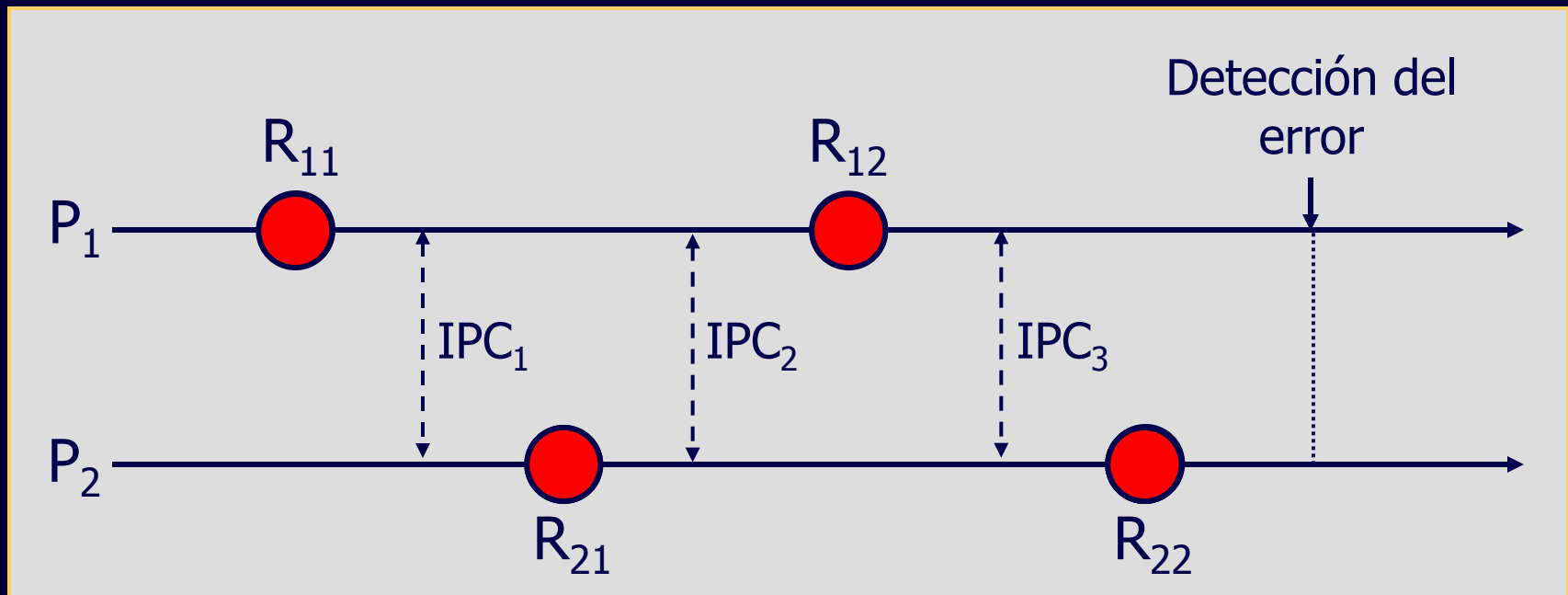
- Continuar tras correcciones selectivas
 - la forma de hacerla es específica de cada sistema
- Depende de una predicción correcta de:
 - los posibles fallos y
 - su situación
- También hay que dejar en un estado seguro al sistema controlado
- Ejemplos:
 - punteros redundantes en estructuras de datos
 - códigos autocorrectores (Hamming)



- Consiste en:
 - retroceder a un estado anterior correcto y
 - ejecutar un segmento de programa alternativo (con otro algoritmo)
- el estado al que se retrocede se llama **punto de recuperación**
- no es necesario averiguar la causa ni la situación del fallo
- sirve para fallos imprevistos
- no puede deshacer los errores que aparecen en el sistema controlado



- Con procesos concurrentes que interaccionan, la recuperación se complica



- Solución: líneas de recuperación (conjuntos consistentes de puntos)



- La reparación automática es difícil y depende del sistema concreto
- Hay dos etapas:
 - **Localización del fallo**
 - mediante técnicas de detección de errores
 - **Reparación del sistema**
 - Los componentes del hardware se pueden cambiar
 - Los componentes del software se *reparan* haciendo una nueva versión
 - En los sistemas sin parada es necesario reemplazar el componente defectuoso sin detener el sistema





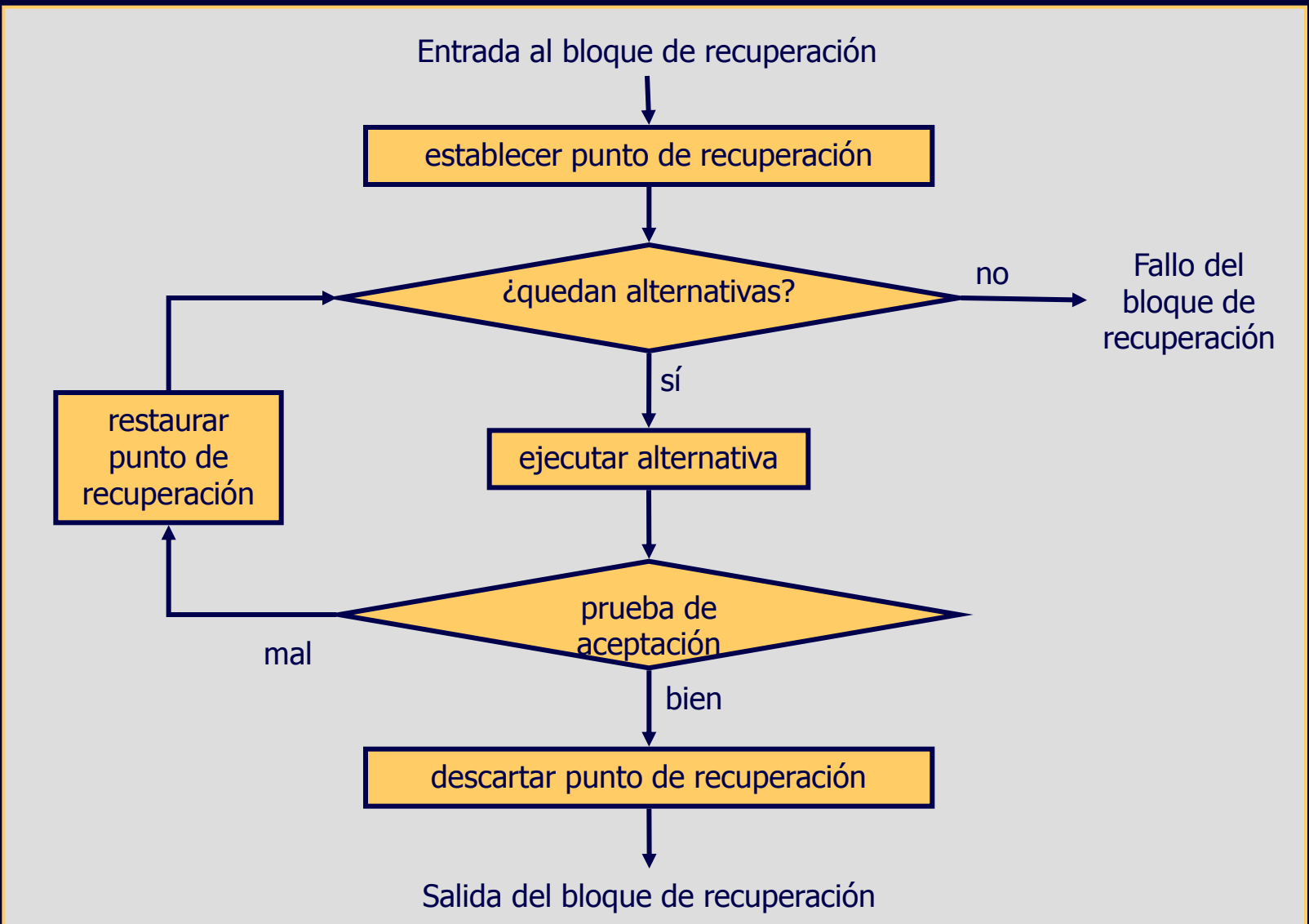
Bloques de recuperación



- Es una técnica de recuperación hacia atrás integrada en el lenguaje de programación
- En un **bloque de recuperación**:
 - su entrada es un **punto de recuperación**
 - a su salida se efectúa una **prueba de aceptación**
 - comprueba si el **módulo primario** del bloque termina en un estado correcto
 - si la prueba de aceptación falla
 - se restaura el estado inicial en el punto de recuperación
 - se ejecuta un **módulo alternativo** del mismo bloque
 - si vuelve a fallar, se siguen intentando alternativas
 - cuando ya no quedan más, el bloque falla y hay que intentar la recuperación en un nivel más alto



Funcionamiento de los bloques de recuperación





```
ensure <condición de aceptación>  
by  
    <módulo primario>  
else by  
    <módulo alternativo>  
else by  
    <módulo alternativo>  
...  
else by  
    <módulo alternativo>  
else error;
```

- Puede haber bloques anidados
 - si falla el bloque interior, se restaura el punto de recuperación del bloque exterior



Ejemplo de codificación en Ada: ecuación diferencial

- Hay un método explícito, que es más rápido, pero cuyo margen de error no es aceptable para algunos tipos de ecuaciones
- El método implícito sirve para todas las ecuaciones, pero es más lento

```
ensure precision <= tolerancia  
by  
    Explicit_Runge_Kutta;  
else by  
    Implicit_Runge_Kutta;  
else error;
```

- Este esquema sirve para todos los casos
- Puede tolerar fallos de programación



- Es fundamental para el buen funcionamiento de los bloques de recuperación
- Hay que buscar un compromiso entre detección exhaustiva de fallos y eficiencia de ejecución
- Se trata de asegurar que el resultado es **aceptable**, no necesariamente **correcto**
- Pero hay que tener cuidado de que no queden errores residuales sin detectar



N versiones

- Redundancia estática
- Diseño
 - algoritmos alternativos
 - proceso guía
- Ejecución
 - múltiples recursos
- Detección de errores
 - votación

Bloques de recuperación

- Redundancia dinámica
- Diseño
 - algoritmos alternativos
 - prueba de aceptación
- Ejecución
 - puntos de recuperación
- Detección de errores
 - prueba de aceptación





Excepciones



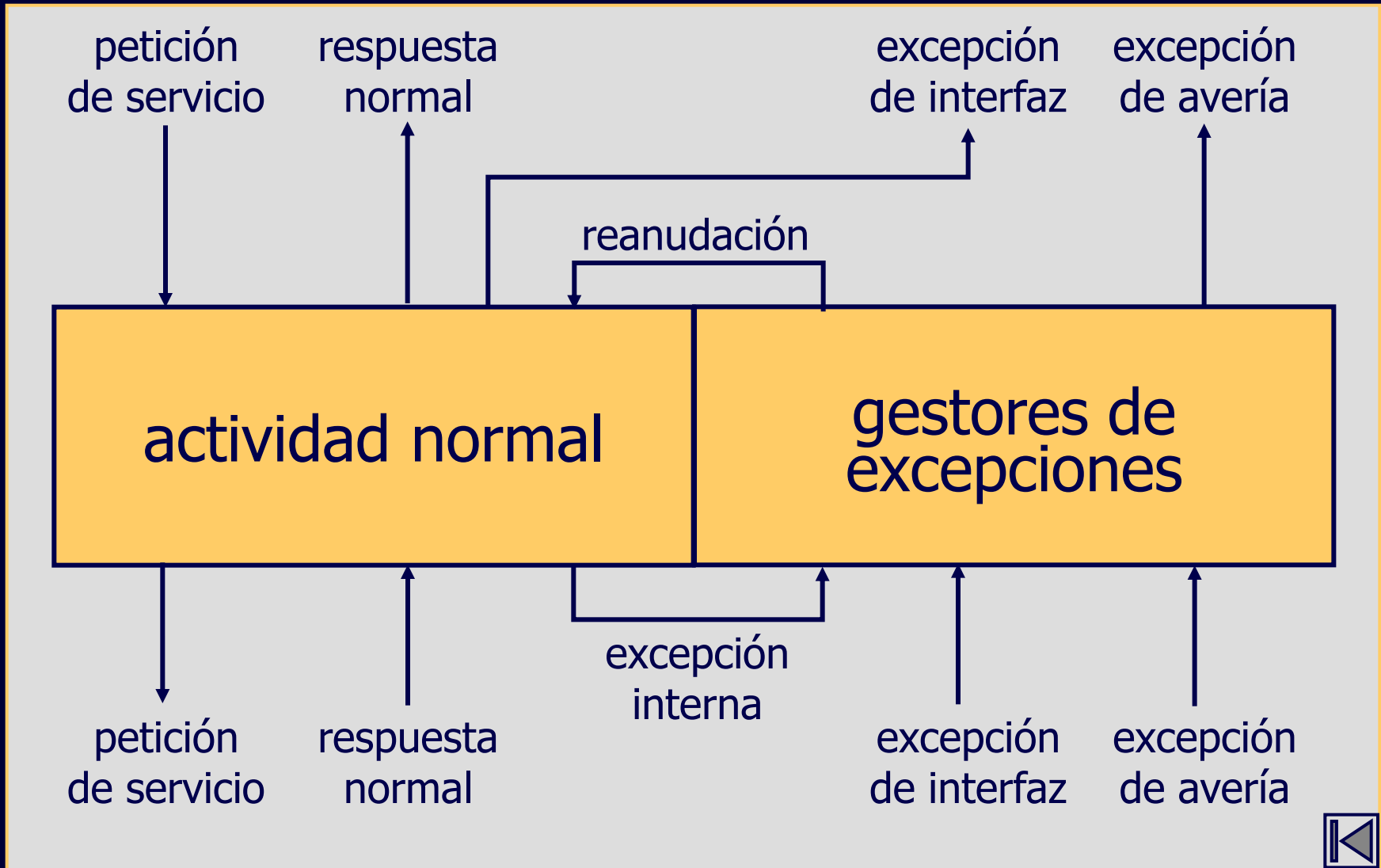
- Una **excepción** es una manifestación de una situación de error
- Cuando se produce un error, se **lanza** la excepción correspondiente en el contexto donde se ha invocado la actividad errónea
- Esto permite **tratar** la excepción en dicho contexto
- Se trata de un mecanismo de recuperación **hacia adelante** de errores
- Pero también se puede utilizar para realizar recuperación hacia atrás



- Hacer frente a situaciones anormales que surjan en el sistema controlado
- Tolerar fallos de diseño del software
- Proporcionar un mecanismo de propósito general para la detección y la corrección de errores



Composición ideal de un sistema tolerante de los fallos





Seguridad, fiabilidad y confiabilidad



- Un sistema es **seguro** si está a salvo de situaciones que puedan causar muertes, heridas, enfermedades (a las personas), o daños en (o pérdida de) los equipos o en el ambiente
 - un accidente (*mishap*) es un suceso imprevisto que puede producir daños inadmisibles
- La seguridad es la probabilidad de que no se produzcan situaciones que puedan conducir a accidentes, independientemente de que se cumpla la especificación o no
- **Fiabilidad**: medida de hasta qué punto un sistema cumple sus especificaciones
 - Un sistema puede ser seguro y no ser fiable
 - y puede ser fiable y no ser seguro



- La confiabilidad (*dependability*) es una propiedad de los sistemas que permite confiar justificadamente en el servicio que proporcionan
- Tiene varios atributos:
 - **disponibilidad**: capacidad de utilización
 - **fiabilidad**: continuidad en la prestación del servicio
 - **seguridad**: no ocurrencia de situaciones catastróficas
 - **confidencialidad**: no ocurrencia de fugas de información no autorizadas
 - **integridad**: no ocurrencia de alteraciones no permitidas de la información
 - **mantenibilidad**: aptitud para soportar reparaciones y evoluciones



- Los medios para conseguirla son:
 - Prevención de fallos
 - Eliminación de fallos
 - Tolerancia de fallos
 - Predicción de fallos
- Los obstáculos que pueden impedirla:
 - Defectos
 - Errores
 - Fallos