UNIVERSIDAD NACIONAL DE LA PLATA

"Identificación y Clasificación de Patrones de Aplicación en Sistemas Embebidos"

Trabajo Final presentado para obtener el grado de Especialista en Ingeniería de Software

Facultad de Informática – Universidad Nacional de La Plata

Alumno: Pedro Ignacio Domingo Martos
Directora: Dra. Alejandra Garrido

Mayo de 2015

Índice

Listado de Figuras

Capitulo 1. Introducción

1.1. Sistemas Embebidos

Un sistema embebido (S.E.) es una plataforma de cómputo (Hardware + Software) con una función específica dentro de un sistema más grande, normalmente con restricciones de ejecución de tiempo real (se necesita que los tiempos de cómputo estén dentro de ciertos límites preestablecidos) [1]. Se denominan “embebidos” a estos sistemas porque son parte de un dispositivo que incluye otros componentes de hardware y/o partes mecánicas y por oposición a los sistemas computacionales de propósito general (p.ej. PC o Tablets), que se diseñan para ser flexibles y para satisfacer las necesidades de una amplia gama de usuarios [2].

La característica principal de los S.E. es que están diseñados para una tarea específica; es por ello que pueden optimizarse para reducir su tamaño y su costo de producción, incrementando su confiabilidad y su performance. Muchos S.E. se diseñan pensando en producción masiva, beneficiándose de la economía de escala (el costo por unidad se reduce al aumentar la cantidad de unidades producidas porque el costo fijo del diseño se distribuye entre todas las unidades producidas)

En el aspecto físico, los S.E. van desde dispositivos portátiles tales como teléfonos celulares, cámaras fotográficas o reproductores MP3, hasta instalaciones fijas tales como controladores de semáforos para tráfico vehicular o maquinas automáticas industriales. También forman parte de los sistemas de control y seguridad en vehículos terrestres y aviónica.

Por otra parte su complejidad va desde sistemas simples implementados con un microcontrolador básico, como puede ser el control remoto de un televisor; hasta sistemas altamente complejos compuestos por varias unidades trabajando en forma concurrente y distribuida, como es el caso de un robot industrial. [3]

1.2. Ingeniería de Software y Sistemas Embebidos

La ingeniería de software en su evolución genera técnicas y herramientas que permiten desarrollar software de manera más sistemática, organizada y predecible, pero a pesar

de ello, su inclusión en el desarrollo de software de sistemas embebidos se hace muy lentamente.

Asi, podemos ver que los problemas asociados al desarrollo de software para S.E inicialmente están centrados en la falta de sistematización del proceso de desarrollo de software, como se describe en Graaf et al [4] (2003):

- Las tecnologías de desarrollo de software no tienen en cuenta las necesidades específicas del desarrollo para S.E. Estas necesidades en general son restricciones de temporización, poca memoria, bajo consumo, y plataformas de hardware definidas e inamovibles.

- Las compañías que desarrollan software para S.E. no venden específicamente software, venden productos tales como teléfonos celulares, reproductores de cd, cámaras fotográficas, etc.; por lo que la ingeniería de software y otros procesos son subprocesos de la ingeniería de sistemas necesaria para desarrollar los productos, es decir, el foco del desarrollo está en el sistema, siendo el software un componente más del mismo.

- No hay una comunicación horizontal entre los equipos de desarrollo que trabajan al mismo nivel de abstracción en distintas partes del sistema. Esta falta de comunicación genera inconvenientes tales como duplicidad de funcionalidad, en la que se implementa la misma funcionalidad en distintas partes del sistema, y recién en la etapa de integración se define qué parte del sistema implementara finalmente la funcionalidad, habiéndose realizado un trabajo innecesario en las otras partes del sistema.

- La ingeniería del sistema está fuertemente centrada en el hardware. En algunos casos el área de software ni siquiera participa en las decisiones de diseño a nivel sistema. La razón de esto suele ser que el hardware tiene retardos en la provisión de componentes y una fuerte dependencia de proveedores externos, por lo que el desarrollo de software suele comenzar cuando el desarrollo de hardware ya se encuentra avanzado, de manera que las propiedades del hardware condicionan el universo de soluciones del software

- En lo que respecta a la ingeniería de requerimientos, en los textos de ingeniería de software la etapa de requerimientos y la de diseño están separadas, en el desarrollo de sistemas embebidos esta separación no es tan simple, ya que puede haber cambios en los requerimientos una vez comenzada la fase de diseño del sistema.

- En los S.E. los requerimientos no funcionales son tan importantes como los funcionales, pero las metodologías de diseño rara vez contemplan los requerimientos no funcionales.

- Las técnicas más comunes de documentación son diagramas “libres” con alguna semejanza con UML, pero sin sistematización, esto da lugar a errores de interpretación de dichos diagramas entre los distintos miembros del equipo de desarrollo. Por otra parte, UML empieza a usarse como estándar para diagramas, pero su aplicación todavía no está muy difundida. Asimismo Los diagramas UML se utilizan principalmente como herramientas de documentación, no como herramientas de modelado del sistema.

- La forma normal de especificar requerimientos es mediante pre y post condiciones expresadas en C o en algún otro tipo de lenguaje. Raramente se utilizan métodos formales, principalmente porque requieren personal altamente especializado, por lo que solo se utilizan en proyectos en los que la seguridad de las personas sea lo más importante.

- La mayoría de los proyectos no comienzan desde cero, sino que un nuevo proyecto surge como una mejora de un proyecto ya existente, lo que implica una reutilización de las especificaciones de requerimientos, el problema es que dichas especificaciones suelen no estar actualizadas, lo que implica que deben ser analizadas nuevamente. Las herramientas para gestión de requerimientos no suelen servir de ayuda.

- Debido a que el software se considera más flexible y tiene menores retardos de provisión, este debe solucionar fallas en la arquitectura del hardware.

- En el proceso de diseño, no suele haber disponible requerimientos de performance, se suele desarrollar los sistemas para que sean “lo más rápido posible”; por otra parte, la descomposición de requerimientos de performance de alto nivel en requerimientos de menor nivel suelen hacerla los desarrolladores basándose en su experiencia previa, lo mismo se aplica para otros requerimientos no funcionales como memoria y consumo.

- La reutilización de elementos se realiza de manera ad-hoc; se reutilizan requerimientos, documentos de diseño y código de proyectos similares mediante copia, debido a que los nuevos proyectos son evolución de proyectos anteriores.

- Hay un salto entre las técnicas de desarrollo disponibles y las técnicas aplicadas en el desarrollo de los proyectos; esto se debe principalmente a que las metodologías de desarrollo están bien definidas a un nivel conceptual, pero su aplicación no es del todo clara en el área de sistemas embebidos. Por otra parte, dichas técnicas de desarrollo requieren profesionales con formación específica en ingeniería de software, los cuales no suelen estar disponibles en el área de sistemas embebidos.

El panorama seis años después nos muestra que si bien el proceso de desarrollo de software para S.E. esta mas sistematizado, gracias a la importación de metodologías que fueron aplicadas con éxito en otros ámbitos de desarrollo de software, el principal problema pasa a ser la falta de profesionales de desarrollo de software para sistemas embebidos con formación específica en ingeniería de software, como se ve en las características del desarrollo de software de sistemas embebidos que se describe en Ebert et al [5] (2009):

- Para el desarrollo de software de S.E. se utilizan prácticas de ingeniería de software centradas en la confiabilidad: Model Driven Design, Model Driven Testing, Métodos Formales, Análisis Estático de Código, Testing Automático,

Evaluación de Madurez de Procesos de Desarrollo (p.ej. CMM), componentes diseñados específicamente para su reutilización (p.ej. librerías gráficas o de procesamiento de señales), etc. El desarrollo de software para sistemas embebidos tiende a ser más formal que en otros ámbitos, fundamentalmente por los requisitos de seguridad y tiempo real asociados. Para ello se utilizan metodologías de aseguramiento de la calidad del software, mediciones de calidad, inspecciones formales y más etapas de prueba que en otros ámbitos de desarrollo de software.

- Los profesionales que desarrollan software para S.E. no suelen concurrir a eventos relacionados específicamente con el desarrollo de software (conferencias, congresos), sino que concurren a eventos específicos de su área de desarrollo (telecomunicaciones, aviónica etc).

- No hay disponibles datos comparativos provenientes de la industria, o los datos no pueden ser aplicados a nuevas metodologías o proyectos. Si bien pueden obtenerse datos de otras aéreas de software, no es el caso del software de S.E. Una razón de esto es que este tipo de software es muy específico del sistema sobre el que se ejecutara, lo que hace difícil hacer estudios comparativos.

- Los desarrolladores de software para S.E. no se consideran a sí mismos como “desarrolladores de software”, sino que se consideran profesionales de otras ramas. Normalmente tienen educación formal en ingeniería electrónica o en telecomunicaciones, y consideran que el desarrollo de software tiene menos status profesional que estas otras formas de ingeniería.

De esta manera, se ve que a pesar de una mayor sistematización en el proceso de desarrollo del software de S.E; aún faltan profesionales dedicados a este ámbito con formación específica en ingeniería de software y técnicas y metodologías orientadas al desarrollo de este tipo de software.

1.3. Patrones

Un “Patrón” en ingeniería de software es una solución general reutilizable que es aplicable a un problema que ocurre comúnmente en un contexto determinado durante la etapa de diseño del software.

Un patrón no se puede transformar directamente en código, sino que es una descripción o “plantilla” sobre cómo se puede resolver un problema, que se puede aplicar en diferentes situaciones. Los patrones son una formalización de buenas prácticas de diseño de software.

Históricamente el concepto de patrones surgió como un concepto arquitectónico, atribuido al arquitecto Christopher Alexander, quien planteo la posibilidad de establecer ideas de diseño arquitectónico como descripciones reutilizables para el diseño de edificios [6].

En el área de software, en 1987 Kent Beck y Ward Cunningham empezaron a aplicar este concepto al desarrollo de software y presentaron sus resultados en la conferencia OOPSLA (Object-Oriented Programming, Systems, Languages & Applications) de ese año. El concepto de patrones en software se popularizo con la publicación del libro “Design Patterns: Elements of Reusable Object-Oriented Software”, publicado en 1994, el cual se considera uno de los libros más influyentes sobre este tema [7].

1.4. Justificación del presente Trabajo

Atento a que de acuerdo a la bibliografía se observa que si bien hay una mayor sistematización del proceso de desarrollo de software para S.E., los principales problemas actuales son la falta de profesionales de este ámbito con formación específica en ingeniería de software y la falta de técnicas o metodologías especificas para el desarrollo de este tipo de software. Es por ello que se propone identificar y clasificar patrones de uso común en el desarrollo de S.E; a fin de generar un catalogo de patrones de aplicación en S.E como forma de contribuir a la formación en técnicas de ingeniería de software para los profesionales que se dedican al desarrollo de software de S.E.

Capitulo 2. Clasificación de Sistemas Embebidos que aplican patrones

La difusión de técnicas de ingeniería de software para el desarrollo de S.E. se realizó primeramente para aquellos sistemas que tienen requerimientos estrictos de tiempo real y para aquellos que tienen requerimientos específicos de seguridad funcional (safety) y confiabilidad, ya que dichos sistemas requieren mucha mayor planificación en su implementación. La aplicación de patrones no es ajena a esta tendencia, por lo que en la bibliografía se encuentran aplicaciones de patrones en S.E. con requerimientos estrictos de tiempo real, seguridad funcional y confiabilidad.

2.1. Sistemas Embebidos con requerimientos de tiempo real (Real Time)

Los S.E. con requerimientos de tiempo real (Real Time Embedded Systems) son aquellos sistemas cuya correcta operación depende tanto de los resultados computacionales del sistema como también del tiempo en el que dichos resultados son generados. Un S.E. de tiempo real “soft” (Soft Real Time Embedded System) es un sistema cuya operación resulta degradada si los resultados no se obtienen dentro de un tiempo especificado. En un S.E. de tiempo real “hard” (Hard Real Time Embedded System) si los resultados no se producen dentro del tiempo especificado, se considera un fallo del sistema [9].

Los S.E. en general suelen especificarse como una máquina de estados finita (FSM – Finite State Machine) en la que se definen los distintos estados del sistema, los eventos que generan una transición entre estados, y la salida del sistema asociada a cada estado. En los S.E. de tiempo real, la especificación de la transición entre estados y la generación de la salida tienen asociada una temporización.

Para la especificación de sistemas como FSM, Dwyer et al [10] definen patrones para la especificación de requerimientos del sistema con el objetivo de aplicar métodos formales (en particular la verificación de estados finitos) a la especificación del sistema. Para ello se modela el sistema como una serie de transiciones con un numero finito de estados y un conjunto de transiciones entre estados asociadas a eventos. Entonces el

patrón para la especificación de propiedades es una descripción generalizada de un requerimiento de ocurrencia en la secuencia de estados/eventos permisibles en ese modelo del sistema.

El patrón para la especificación de propiedades describe entonces la estructura básica de un aspecto del comportamiento del sistema y provee una expresión de este comportamiento en un formalismo. Para ello para cada requerimiento se define un "Alcance", que determina en qué ámbito se aplica dicho requerimiento, y seguidamente se define el requerimiento utilizando alguno de los patrones definidos. Los posibles alcances se describen en la Tabla 1 y los patrones para la definición del requerimiento en la Tabla 2.

Alcance (scope)	Descripción
Global (globally)	Durante toda la ejecución del programa
Antes (before)	Antes de la ocurrencia del estado/evento P
Después (after)	Después de la ocurrencia del estado/evento P
Entre (between P and Q)	En algún momento desde la ocurrencia del estado/evento P hasta la ocurrencia del estado/evento Q
Después-hasta (After P until Q)	Similar al anterior pero contempla la posibilidad que el estado/evento Q no ocurra.

Tabla 1: Posibles alcances de un requerimiento

Categoría	Patrón	Descripción
Ocurrencia	Universalidad	Un determinado estado/evento debe suceder durante todo el alcance
	Existencia	Un determinado estado/evento debe ocurrir dentro del alcance
	Ausencia	Un determinado estado/evento no debe ocurrir dentro del alcance
	Existencia Acotada	Un determinado estado/evento debe ocurrir k veces dentro del alcance; hay dos variantes: que ocurra como máximo k veces; y que ocurra como mínimo k veces
Orden	Precedencia	Un estado/evento P debe estar siempre precedido por un estado/evento Q dentro del alcance
	Respuesta	Un estado/evento P debe estar siempre seguido por un estado/evento Q dentro del alcance
	Cadena de Precedencia	Una secuencia de estados/eventos P1…Pn debe estar siempre precedida por una secuencia de estados/eventos Q1…Qn dentro del alcance
	Cadena de Respuesta	Una secuencia de estados/eventos P1…Pn debe estar siempre seguida por una secuencia de estados/eventos Q1…Qn dentro del alcance

Tabla 2: Patrones para la definición de requerimientos

Si bien de esta manera se establecen patrones para la especificación de requerimientos en un S.E. modelado como FSM, estos patrones no incluyen

información de temporización, lo cual es necesario en S.E. con requerimientos de tiempo real.

Es por ello que en Konrad et al [9] se definen patrones para la especificación de los requerimientos de tiempo real de un sistema mediante la descripción de los mismos utilizando una gramática estructurada. Esto se realiza integrando el requerimiento no funcional de temporización como parte de la descripción del patrón y clasificando los patrones en tres categorías de acuerdo a si el requisito de tiempo real está asociado a la duración de un evento, a la periodicidad de un evento, o al orden en que deben suceder dos eventos. En la Tabla 3 se describen estos patrones que incluyen requerimientos de temporización.

Categoría	Patrón	Descripción
Duración	Mínima Duración	Describe la mínima cantidad de tiempo que un sistema debe permanecer en determinado estado una vez que se cumple determinada condición Ej: “Si el quemador se apaga, la llave de paso de gas debe permanecer cerrada por lo menos 120 segundos antes de iniciarse el reencendido del quemador”.
	Máxima Duración	Describe que el sistema debe permanecer en determinado estado una cantidad menor de tiempo que el tiempo especificado. Ej: “Durante el encendido, si no hay detección de llama en el quemador, la válvula de paso de gas debe permanecer abierta como máximo 10 segundos”
Periodicidad	Recurrencia Acotada	Describe la cantidad de tiempo en que un determinado estado del sistema debe suceder al menos una vez. Ej: “La detección de llama en el quemador debe verificarse por lo menos cada 100 milisegundos”
Orden	Respuesta Acotada	Restringe la máxima cantidad de tiempo que pasa desde que una condición se cumple hasta que otra condición se cumple. Ej: “Si se detecta ausencia de llama en el quemador, la llave de paso del gas debe cerrarse en menos de 0.5 segundos”
	Invariancia Acotada	Especifica la mínima cantidad de tiempo que el sistema debe permanecer en un estado una vez que se satisface la condición de otro estado. Ej: “Si falla el encendido del quemador, la secuencia de encendido queda anulada durante 60 segundos”

Tabla 3: Patrones para la especificación de requerimientos que incluyen temporización

La forma de descripción de estos patrones se denomina “Ingles Estructurado” (Estructured English), de manera de obtener una representación en lenguaje natural de las propiedades de tiempo real del sistema; ya que el autor considera que esta representación es más accesible para personas sin formación en métodos formales

que utilizando otras notaciones tales como las lógicas temporales MTL (Metric Temporal Logic) [11]; TCTL (Timed Computational Tree Logic) [12] o RTGIL (Real Time Graphical Interval Logic) [13].

La gramática del Ingles Estructurado permite crear una representación en lenguaje natural de la siguiente manera: inicialmente se establece el alcance de la propiedad (utilizando la misma definición de alcance que Dwyer et al [10]), seguido del tipo (cuantitativo o de tiempo real). Después se selecciona la categoría (duración, periódica, orden de tiempo real) para propiedades de tiempo real y (ocurrencia, orden) para propiedades cualitativas. La oración en Ingles Estructurado se construye entonces de acuerdo al patrón correspondiente. Cabe aclarar que los términos “precede” (“precede”) y “sucede” (“succeed”) denotan pasado y futuro inmediatos respectivamente, mientras que “Sucedido previamente” (“held previously”) y “Sucedido eventualmente” (“hold eventually”) denotan pasado y futuro no necesariamente inmediatos. Así, por ejemplo, para especificar que un evento A genera una transición a un estado S, se diría “El evento A *precede* al estado S”, mientras que para especificar que el evento A sucedió en algún momento previamente al estado S, se diría “El evento A, *sucedido previamente* al estado S…” Del mismo modo, para especificar que un estado S genera como salida un evento A, se diría “El evento A sucede al estado S”, mientras que para especificar que el evento A sucederá en algún futuro a partir del estado S, se diría “El evento A, *sucedido eventualmente* a partir del estado S…” En la Tabla 4 se muestra la representación de esta gramática en ingles. Finalmente, cabe aclarar que en Konrad et al [9] se agrega la descripción en Ingles Estructurado de los patrones definidos en Dwyer et al [10].

Start	1: property	::=	*scope* ", " *specification* "."
Scope	2: scope	::=	"Globally" | "Before " R | "After " Q | "Between " Q " and " R | "After " Q " until " R
General	3: specification	::=	*qualitativeType* | *realtimeType*
Qualitative	4: qualitativeType	::=	*occurrenceCategory* | *orderCategory*
	5: occurrenceCategory	::=	*absencePattern* | *universalityPattern* | *existencePattern* | *boundedExistencePattern*
	6: absencePattern	::=	"it is never the case that " P " holds"
	7: universalityPattern	::=	"it is always the case that " P " holds"
	8: existencePattern	::=	P " eventually holds"
	9: boundedExistencePattern	::=	"transitions to states in which " P " holds occur at most twice"
	10: orderCategory	::=	"it is always the case that if " P " holds" (*precedencePattern* | *precedenceChainPattern1-2* | *precedenceChainPattern2-1* | *responsePattern* | *responseChainPattern1-2* | *responseChainPattern2-1* | *constrainedChainPattern1-2*)
	11: precedencePattern	::=	", then " S " previously held"
	12: precedenceChainPattern1-2	::=	" and is succeeded by " S ", then " T " previously held"
	13: precedenceChainPattern2-1	::=	", then " S " previously held and was preceded by " T
	14: responsePattern	::=	", then " S " eventually holds"
	15: responseChainPattern1-2	::=	", then " S " eventually holds and is succeeded by " T
	16: responseChainPattern2-1	::=	" and is succeeded by " S ", then " T " eventually holds after " S
	17: constrainedChainPattern1-2	::=	", then " S " eventually holds and is succeeded by " T ", where " Z " does not hold between " S " and " T
Real-time	18: realtimeType	::=	"it is always the case that " (*durationCategory* | *periodicCategory* | *realtimeOrderCategory*)
	19: durationCategory	::=	"once " P " becomes satisfied, it holds for " (*minDurationPattern* | *maxDurationPattern*)
	20: minDurationPattern	::=	"at least " c " time unit(s)"
	21: maxDurationPattern	::=	"less than " c " time unit(s)"
	22: periodicCategory	::=	P " holds " *boundedRecurrencePattern*
	23: boundedRecurrencePattern	::=	"at least every " c " time unit(s)"
	24: realtimeOrderCategory	::=	"if " P " holds, then " S " holds " (*boundedResponsePattern* | *boundedInvariancePattern*)
	25: boundedResponsePattern	::=	"after at most " c " time unit(s)"
	26: boundedInvariancePattern	::=	"for at least " c " time unit(s)"

Tabla 4: Patrones para la especificación de requerimientos de tiempo real mediante gramática estructurada

Por otra parte, los S.E. con requerimientos de tiempo real tienen distintos patrones para definir la arquitectura de su implementación, tal como se describen en el libro de Douglas [14]. En este trabajo el autor describe patrones de aplicación en S.E. con requerimientos de tiempo real utilizando la misma estructura que [7], es decir, sin agregar requerimientos no funcionales en la descripción del patrón, y los clasifica en tres Áreas, que se describen en la Tabla 5

Área de aplicación del Patrón	Descripción
Patrones para arquitectura de subsistemas y componentes	Patrones aplicables a la organización y a la arquitectura del sistema
Patrones para concurrencia	Patrones para gestión de recursos que deben ser protegidos de accesos simultáneos
Patrones para gestión de la memoria	Patrones para la gestión eficiente de la memoria

Tabla 5: Áreas de aplicación de los patrones arquitecturales de Douglas [14]

Cabe aclarar que algunos de estos patrones son repetición o especialización de otros patrones, como por ejemplo “Recursive Containment Pattern”, que es similar al patrón “Facade” de [7]

2.2. Sistemas Embebidos con requerimientos de seguridad funcional y confiabilidad (safety and reliability)

Los S.E. con requerimientos de seguridad funcional y confiabilidad son sistemas cuya falla puede tener graves consecuencias económicas o daños a la vida humana. Es por ello que su confiabilidad es crítica y el diseño del sistema debe incorporar esta característica. Esto normalmente conduce a criterios conservadores en el diseño, donde se aplican técnicas probadas y evaluadas en vez de nuevas técnicas de diseño, las cuales podrían introducir nuevos modos de falla en el sistema [11].

La seguridad funcional está asociada con la posibilidad que un mal funcionamiento del sistema produzca daños, mientras que la confiabilidad está asociada con la posibilidad que el sistema permanezca funcionando correctamente a pesar de producirse un malfuncionamiento en el mismo. Para aumentar la seguridad funcional y/o la confiabilidad es necesario implementar alguna forma de redundancia. La seguridad funcional y la confiabilidad suelen ser requerimientos opuestos, ya que frente a un malfuncionamiento del sistema la seguridad funcional implica conducirlo a un “estado seguro” (safety state) que no represente riesgos para el operador o el usuario, pero en este estado normalmente la mayor parte de la funcionalidad del sistema esta deshabilitada; lo cual es opuesto a la confiabilidad, que requiere que la funcionalidad este continuamente disponible aun en presencia de un malfuncionamiento.

Desde el punto de vista de la seguridad funcional y la confiabilidad, se produce un *fallo* en un sistema debido a que el mismo tiene algún tipo de *error* o debido a que algún aspecto del sistema *falla*. Un *error* es una falla sistemática que puede deberse a requerimientos incorrectos, un mal diseño, o una mala implementación. Un *error* está siempre presente en el sistema, aunque puede no ser visible, cuando un *error* es visible, se dice que se *manifiesta*. Una *falla* es diferente, debido a que ocurre en algún momento durante el funcionamiento del sistema y lo conduce a un estado anormal; es decir, el sistema funciona correctamente hasta que algo en él cambia, si el cambio es

irreversible, por ejemplo si algo se rompe, es una *falla persistente*. Por otra parte, si la falla es reversible y es posible volver a un estado normal, por ejemplo hay un error en un dato en la memoria y este es corregido porque la memoria implementa algún algoritmo de corrección de errores (ECC), se denomina una *falla volátil*.

Esta división de los *fallos* de un sistema en "*errores"* y "*fallas"* es importante debido a que las técnicas de mitigación son distintas: las *fallas* se pueden mitigar mediante *redundancia homogénea*, es decir, múltiples copias de un mismo elemento. Pero si el *fallo* es un *error*, aplicar esta técnica es ineficaz debido a que todas las copias fallaran de la misma manera. En este caso se aplica la *redundancia heterogénea*, que es implementar la misma funcionalidad de manera diferente, utilizando otro código fuente generado a partir de los mismos requerimientos.

En los patrones para este tipo de S.E. suelen presentarse tres elementos: el *manifestador*, que es el elemento en el que se produce el fallo; el *identificador*, que es el elemento que puede identificar cuando ha ocurrido un falla o cuando un error se manifiesta en el sistema; y el *reductor*, que es el elemento que hace que el fallo del sistema sea menos severo o menos probable [16]. Asimismo estos patrones se aplican sobre el canal de procesamiento, o sobre los datos que se procesan. En este contexto el canal de procesamiento es un subsistema que adquiere algún tipo de dato, realiza un procesamiento o transformación del mismo, o realiza una toma de decisión en base al dato; y genera una salida, que puede ser el dato transformado o una acción sobre otra parte del sistema. Para más detalles, ver el patrón "Canal" en la sección 3.1.4.

En la bibliografía se encuentran patrones relacionados con la seguridad funcional y la confiabilidad, debido a que permiten la reutilización de soluciones ya probadas, y por ende, más confiables, aumentando la confiabilidad total del sistema al que se aplican los patrones. En el trabajo de Armoush et al [17] se propone una plantilla para patrones que contempla los requerimientos no funcionales de Confiabilidad, Seguridad Funcional, Costo, Modificabilidad y Tiempo de Ejecución. En la Tabla 6 se explican cada uno de estos requerimientos.

Requerimiento no funcional	Descripción
Confiabilidad	Describe en forma cuantitativa la mejora en la confiabilidad esperable al aplicar el patrón
Seguridad Funcional	Se indica el nivel de seguridad funcional (SIL), de acuerdo a lo definido en el estándar IEC 61508 [18] que se espera alcanzar
Costo	Contempla los costos por unidad del producto final y los costos de implementar el patrón.
Modificabilidad	Describe que tanto se puede modificar o cambiar el sistema al que se le ha aplicado el patrón.
Tiempo de Ejecución	Estima la modificación en el tiempo de ejecución promedio y de peor caso del sistema al aplicar el patrón

Tabla 6: Requerimientos no funcionales en los patrones descriptos en Armoush et al [17]

Por otra parte, en el libro de Douglas [14] se describen patrones de seguridad funcional y confiabilidad siguiendo el esquema de [7], es decir, sin incluir requerimientos no funcionales en la descripción del patrón.

2.3. Descripción de los patrones

Para la descripción de los patrones, se siguió el esquema presentado en [7], agregándose los campos “Características Particulares”, donde se dan aclaraciones y/o se especifican requerimientos particulares de tiempo real o seguridad; “Fuente”, donde se hace referencia al artículo o trabajo que presenta el patrón; y “Área de aplicación”, que especifica el área de aplicación del patrón. De esta manera, cada patrón se caracteriza de la siguiente manera:

Nombre: Es la forma de describir el problema, su solución y consecuencias en una o dos palabras.

Fuente: referencia (Artículo, Trabajo, Libro, etc.) en el que se presenta el patrón

Área de Aplicación: Área específica del patrón (tiempo real, seguridad funcional, hardware, etc.)

Problema: Describe cuando aplicar el patrón, explica el problema y su contexto.

Solución: Describe los elementos que componen la solución y la relación entre ellos.

Consecuencias: Describe las relaciones de compromiso al aplicar el patrón, permitiendo un análisis comparativo entre distintos patrones aplicables.

Características particulares: Describe los requerimientos de tiempo real o de seguridad involucrados en la aplicación del patrón.

Capitulo 3. Patrones de S.E. en general

3.1. Arquitectura del Software de S.E.

3.1.1.Arquitectura en capas

Fuente:
Real Time Design Patterns: Robust Escalable Architecture for Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:
Arquitectura de componentes y subsistemas

Problema:
Se implementa un sistema para dar solución a un problema determinado dentro de un Dominio [*], pero su implementación se realiza mediante elementos de otros Dominios, por lo que es necesario establecer una jerarquía que relacione los distintos Dominios involucrados en la resolución del problema.

Solución:
Este patrón organiza los distintos Dominios de la implementación de un sistema en una jerarquía basada en su *Nivel de Abstracción*. Los conceptos abstractos de un Dominio determinado son implementados mediante conceptos más concretos que pertenecen a otro Dominio.

Consecuencias:
Aplicar este patrón permite trabajar con conceptos altamente abstractos y pertenecientes al Dominio de la aplicación, implementándolos mediante conceptos más concretos. Las arquitecturas implementadas de esta manera son altamente portables entre plataformas ya que las capas superiores son mas especificas de la aplicación y las capas inferiores son mas especificas de la plataforma de cómputo donde se implementa el sistema. La portabilidad se da en ambas direcciones, ya que las aplicaciones son mas portables debido a que las capas inferiores de la arquitectura pueden reemplazarse fácilmente por otras capas pertenecientes a otra plataforma de computo (ej. Pasar de una PC de escritorio a un dispositivo móvil como celular o tablet); y por otra parte las capas superiores pueden ser reemplazadas para permitir la ejecución de otra aplicación sobre la misma plataforma de computo.

[*] Un **Dominio** es un campo de estudio que define un conjunto de requerimientos comunes, terminología y funcionalidad para cualquier programa de software construido para resolver un problema. (Wikipedia, noviembre de 2014)

Características Particulares:
Las relaciones jerárquicas deben ser en dirección descendente, es decir, las capas superiores deben depender de las capas inferiores, pero las capas inferiores más concretas no deben depender de las capas superiores más abstractas.

Un patrón relacionado es el de “Arquitectura de Cinco Capas”, que es un caso particular de este patrón

3.1.2.Arquitectura de 5 capas

Fuente:

Real Time Design Patterns: Robust Escalable Architecture for Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:

Arquitectura de componentes y subsistemas

Problema:

En S.E. pequeños y medianos puede ser necesario que una aplicación pueda ser portable a otra plataforma (p.ej. de un microcontrolador a otro), o puede ser necesario que una plataforma tenga que tener la capacidad de ejecutar distintas aplicaciones

Solución:

La arquitectura del sistema se define mediante un conjunto específico de cinco capas, cuya relación jerárquica se muestra en la Figura 1

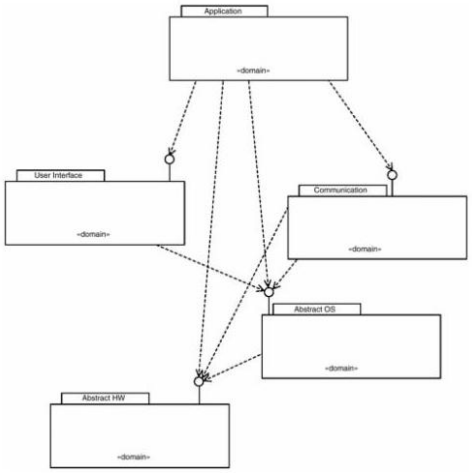

Fig. 1: Arquitectura de 5 Capas

Las cinco capas y Dominios son los siguientes:

- Aplicación: Contiene las clases especificas de la aplicación, utilizando el vocabulario y las relaciones de la misma.
- Interface de Usuario: Contiene clases especificas de la U.I: ventanas, barras de desplazamiento, imágenes, etc.
- Comunicación: contiene clases necesarias para transportar datos entre los objetos, normalmente se subdivide en dos subdominios: *Middleware* (que contiene clases para la gestión de la comunicación entre objetos, como ser Message, Proxy, etc.) y *Data Transport*, que contiene clases para trabajar sobre redes, gestionar la fragmentación de paquetes, implementar transporte confiable sobre medios no confiables, crear sesiones, etc.
- OS Abstracto: Utiliza Adapters que aíslan al sistema de la estructura específica del Sistema Operativo, incluye clases para la gestión de procesos y memoria.
- Hardware Abstracto: Contiene clases que representan dispositivos y su interface

Consecuencias:

Al ser un caso particular del patrón “Arquitectura en Capas”, comparte sus consecuencias, con la aclaración de que al tener pocas capas, puede no ser adecuado como descomposición de Dominios para sistemas complejos.

Características Particulares:

Las mismas que para el patrón “Arquitectura en Capas”

3.1.3.MicroKernel

Fuente:

Real Time Design Patterns: Robust Escalable Architecture for Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:

Arquitectura de componentes y subsistemas

Problema:

Hay subsistemas que por sus características de complejidad y confiabilidad es necesario poder reutilizarlos en una amplia variedad de contextos y plataformas de cómputo.

Solución:

Un subsistema puede implementarse a partir de un conjunto básico de servicios de tal manera que el desarrollador puede elegir cuales servicios utilizar en una aplicación dada. Esta modularidad hace al subsistema altamente reutilizable, ya que provee la capacidad de ser configurado al momento de integrarlo con la aplicación. Un ejemplo muy común de esta clase de subsistema es un RTOS (Real Time Operating System). Estos subsistemas implementan una funcionalidad básica (crear tareas, eliminarlas, asignar memoria, proveer comunicación entre las tareas, etc.), y el desarrollador puede configurar funcionalidad adicional para proveer más servicios, tales como gestión de archivos en distintos medios de almacenamiento (pendrive, tarjeta SD), compresión de datos, comunicación en red (TCP/IP), etc. De esta manera, el RTOS es utilizable en una mayor variedad de aplicaciones y plataformas de cómputo, desde sistemas muy restringidos en memoria a sistemas multiprocesador conectados en red para cómputo distribuido. En la Figura 2 se ve la estructura de este patrón

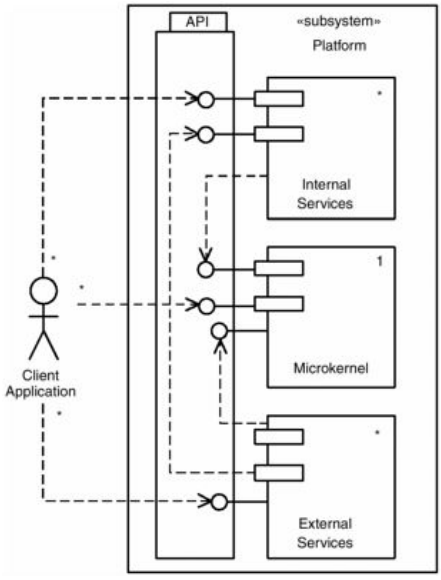

Fig. 2: Microkernel

- API: La Application Program Interface tiene las interfaces de los componentes *Microkernel*, *Servicios Internos* y *Servicios Externos*, puede haber más de una interface por subsistema
- Cliente: Es quien utiliza los servicios provistos por el subsistema, generalmente es la parte del sistema general que se diseña de acuerdo a la aplicación.
- Servicios Externos: Es un componente que provee un conjunto opcional de servicios, los cuales están disponibles al *Cliente* a través de un conjunto de interfaces. Es común que haya varios subsistemas de *Servicios Externos*, que proveen diferente funcionalidad. Este componente es normalmente diseñado por un programador de sistema para aumentar la funcionalidad provista originariamente por la *Plataforma*
- Servicios Internos: Es un componente que provee un conjunto opcional de servicios, los cuales están disponibles al *Cliente* a través de un conjunto de interfaces. Es común que haya varios subsistemas de *Servicios Internos*, que proveen diferente funcionalidad. Este componente es normalmente provisto por el diseñador de la *Plataforma*
- Microkernel: Este componente provee el conjunto mínimo de servicios del subsistema, los cuales están disponibles al *Cliente* a través de un conjunto de interfaces. Es común que haya diferentes interfaces que proveen diferente funcionalidad. Los componentes de *Servicios Internos y Externos* suelen utilizar los servicios provistos por el *Microkernel*
- Plataforma: Es el sistema reutilizable construido a partir de varios componentes que provee un conjunto de interfaces a través de la API

Consecuencias:

Este patrón permite generar un subsistema escalable en el cual un conjunto opcional de servicios puede ser agregado al sistema durante la integración del mismo. Esto permite que el subsistema sea configurado de manera óptima respecto del entorno de ejecución de la aplicación. Para el agregado de componentes durante la ejecución, es más común la utilización del patrón “Componentes”

Características Particulares:

El Microkernel provee los servicios básicos a partir de los cuales los Servicios Internos y Externos proveen servicios más específicos y/o complejos. La provisión de distintas

interfaces permite realizar relaciones de compromiso entre funcionalidad provista, tamaño de memoria necesario, velocidad de ejecución y consumo del sistema

3.1.4.Canal

Fuente:

Real Time Design Patterns: Robust Escalable Architecture for Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:

Arquitectura de componentes y subsistemas

Problema:

Muchos algoritmos procesan un flujo de datos continuo, aplicando el mismo conjunto de operaciones a cada dato. Para mejorar la eficiencia y confiabilidad de un sistema es necesario poder realizar el procesamiento en paralelo de estos datos y/o poder tener redundancia en el procesamiento a través de múltiples instancias del algoritmo actuando sobre un mismo dato.

Solución:

Un Canal puede pensarse como una cadena que secuencialmente transforma un dato de entrada en un dato de salida. Los elementos internos del canal trabajan sobre el flujo de datos como en una cadena de montaje de una fábrica: en forma secuencial, cada elemento realiza una operación simple sobre su dato de entrada, generando un dato de salida que es la entrada del siguiente elemento en la secuencia. Esta arquitectura en sistemas multicore permite tener varias instancias ejecutándose en paralelo, mejorando la capacidad de procesamiento, y/o también se pueden tener varias instancias “canales” procesando el mismo dato simultáneamente, aumentando la confiabilidad del sistema. En la Figura 3 se ve un diagrama de este Patrón

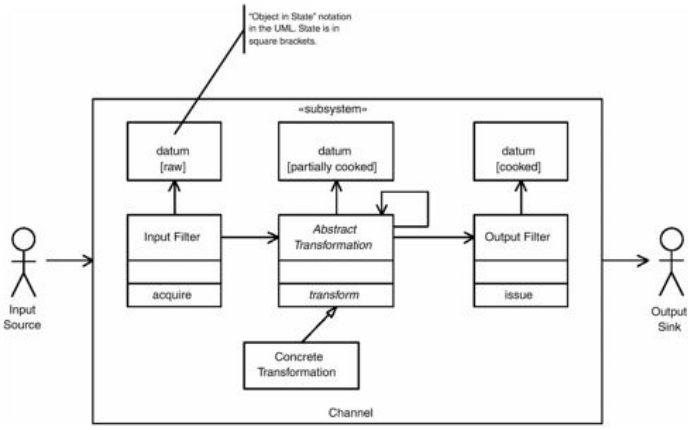

Fig. 3: Canal

Los datos se procesan de manera secuencial desde “Input Source”, que genera los datos a procesar, pasa a través de la etapa “Input Filtering”, luego puede pasar por una o varias etapas “Transformational Processing”, hasta el “Output Filtering” y llegar al “Output Sink”, quien consume los datos procesados. Cada etapa representa un objeto con sus propiedades individuales. El dato en sí mismo es un objeto que sufre un cambio de estado en cada transformación; comienza en el estado “Crudo” (raw) y es gradualmente cocinado (procesado) hasta ser “cocido” (Roasted).

Descripción de los elementos:

- Abstract Transformation: Esta clase provee una interface genérica para las transformaciones que puede sufrir un dato
- Channel: Es un objeto que representa el subsistema completo, y mediante relaciones de composición incluye todos los otros elementos del patrón. La ventaja de usar el canal como un objeto compuesto es que permite tratarlo como una entidad individual y replicarla para mejorar la capacidad de procesamiento o la confiabilidad.
- Concrete Transformation: Es una clase concreta (instanciable) que implementa la interface de *Abstract Transformation*. Su operación *Transform()* es específica para la etapa del algoritmo correspondiente.
- Datum: Es el objeto dato que cambiara de estado en cada sucesiva transformación.
- Input Filter: Este objeto recibe el dato y realiza un procesamiento básico inicial.
- Output Filter: Este objeto entrega el dato procesado al *Output Sink*, normalmente realiza le aplica al dato un formato comprensible por el *Output Sink*.

Consecuencias:

Este patrón es apropiado para la transformación secuencial de datos. Simplifica la implementación de algoritmos que se pueden descomponer en una serie de pasos que se aplican en elementos aislados de un flujo de datos. Se pueden crear varias instancias de un mismo canal para mejorar la capacidad de procesamiento y/o la confiabilidad. Asimismo esta arquitectura puede adaptarse para manejar múltiples elementos (datos) en paralelo, aun si los mismos están en distintos estados de procesamiento (cocción).

Características Particulares:

Hay patrones asociados para mejorar la confiabilidad del sistema, por ejemplo Monitor-Actuador, que implementa dos canales, uno que realiza el procesamiento, y otro que monitorea al primero. El patrón antes mencionados se describe en la parte de Patrones para S.E. con requerimientos de Seguridad Funcional y Confiabilidad.

3.1.5.Control Jerárquico

Fuente:

Real Time Design Patterns: Robust Escalable Architecture for Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:

Arquitectura de componentes y subsistemas

Problema:

En algunos subsistemas es necesario separar la configuración respecto de la funcionalidad del mismo mediante interfaces separadas. Por ejemplo, en un objeto encargado de la compresión de video es necesario separar la configuración (tipo de compresión, bit rate, formato de salida, etc.), respecto de la funcionalidad específica (compresión, descompresión, transcodificación, etc.)

Solución:

Este patrón implementa dos tipos de interfaces, una de control que monitorea y controla como se logra el comportamiento, y una interface funcional, que provee los servicios necesarios. La estructura del patrón se muestra en la Figura 4

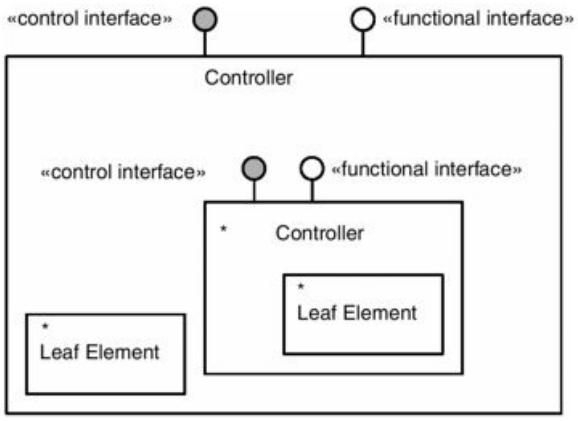

Fig. 4: Control Jerárquico

Este patrón se implementa de manera jerárquica mediante relaciones de composición. El Controller realiza la implementación de sus servicios mediante la delegación y control de objetos internos a él, que a su vez también pueden ser controllers con menor nivel de abstracción.

Los elementos de este patrón son:

- Control Interface: Provee una manera de configurar como se implementara la funcionalidad. Normalmente esta interface no es invocada por los mismos clientes que invocan la interface de funcionalidad.
- Controller: Este objeto provee dos tipos de interfaces a sus clientes: la interface de control y la interface de funcionalidad. Su implementación es a través de delegación, que pueden ser otros *Controllers*.
- Functional Interface: Provee los servicios del subsistema, los cuales son configurados a través de la *Control Interface*
- Leaf Element: Es una clase que no es otro *Controller*, en general no tienen relaciones de composición con otros objetos, pero pueden tener relación con otros objetos. Realizan parte de los servicios provistos por la interface de funcionalidad.

Consecuencias:

Es un patrón aplicable cuando es necesario que el subsistema sea altamente configurable y/o el cliente de configuración es distinto del cliente de funcionalidad, por lo que conviene separar las interfaces de configuración y funcionalidad.

Características Particulares:

Este patrón es una especialización del patrón “Facade” de [7], con la particularidad que la interface se descompone en Control y Funcionalidad

3.1.6.Maquina Virtual

Fuente:

Real Time Design Patterns: Robust Escalable Architecture for Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:

Arquitectura de componentes y subsistemas

Problema:

Para algunas aplicaciones muy sensibles (p.ej. pago electrónico), es necesario que puedan ejecutarse en plataformas muy distintas (PC con S.O. Windows, Linux y OSX; Smartphones con S.O.Android; y/o smartcards en celulares básicos). Realizar una implementación para cada plataforma reduce la confiabilidad, por lo que es un requerimiento realizar una única implementación que funcione en todas las plataformas.

Solución:

Las aplicaciones escritas para usar una maquina virtual se ejecutan en un hardware abstracto, que interpreta las instrucciones para la maquina virtual y las convierte en instrucciones ejecutables por la plataforma de hardware real. De esta manera se implementa una infraestructura capaz de ejecutar aplicaciones que es altamente portable entre plataformas muy distintas en cuanto a sus capacidades de procesamiento y memoria.

Consecuencias:

La alta portabilidad de la aplicación se logra porque frente a una nueva plataforma de computo, solo se necesita implementar la maquina virtual. Por otra parte, al implementar una aplicación para una plataforma, pasa a estar disponible para todas las otras plataformas en las que se pueda ejecutar la maquina virtual.

Características Particulares:

Este patrón favorece la portabilidad a expensas de la eficiencia en tiempo de ejecución. El ejemplo más popular al respecto es JAVA. Al escribir aplicaciones en Java, las mismas se pueden ejecutar en cualquier entorno en el que haya implementada una maquina virtual Java. Tales entornos van desde una PC multicore con varios Gigabytes de RAM corriendo un S.O. de alto nivel como Windows o Linux, hasta una tarjeta SIM de celular, que es una smartcard con un procesador de 32 bits, corriendo un S.O. propietario del fabricante y con menos de 1 Megabyte de RAM.

3.1.7.Arquitectura Basada en Componentes

Fuente:

Real Time Design Patterns: Robust Escalable Architecture for Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:

Arquitectura de componentes y subsistemas

Problema:

En sistemas complejos que requieren una alta mantenibilidad, es necesario poder reemplazar partes del sistema para corrección de errores y/o aumento de la funcionalidad sin tener que recompilar todo el sistema. A su vez, el lenguaje de programación puede no ser el mismo para todos los elementos del sistema, por lo que su interrelación debe ser independiente de la semántica del lenguaje de programación.

Solución:

Un componente es un elemento de código ejecutable que es la unidad mínima reemplazable de software dentro del sistema. Los componentes tienen un fuerte encapsulamiento y una interface bien definida que es independiente del lenguaje de

programación usado para su implementación. Un sistema con una arquitectura basada en componentes utiliza estos elementos como las unidades básicas del mismo, con la intención de que el sistema pueda ser mantenido mediante el reemplazo de componentes individuales. La estructura de este patrón se ve en la Figura 5

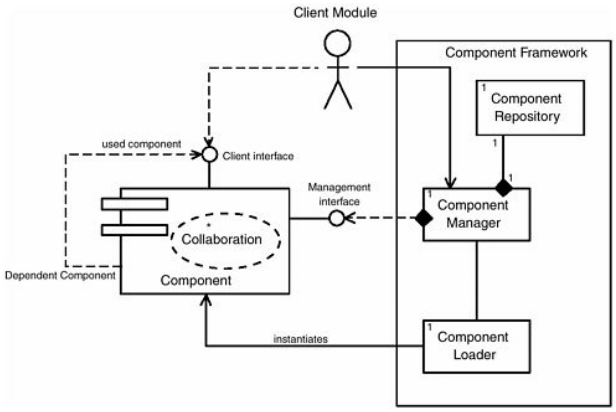

Fig. 5: Arquitectura basada en Componentes

Los elementos de este patrón son:

- Client Interface: Esta interface define los servicios provistos por el componente. Debe ser totalmente opaca, es decir, no debe mostrar nada de la implementación interna, para poder sustituir el componente por otro con la misma interface.
- Colaboration: Es el conjunto de objetos que realizan la implementación de la funcionalidad del componente.
- Component: Es el objeto que mediante relaciones de composición con uno o varios objetos *Colaboration* y provee la interface del componente.
- Component Framework: Es el responsable de la carga y gestión del componente durante su ciclo de vida en tiempo de ejecución. Está integrado por el *Component Loader, Component Manager* y el *Component Repository.*
- Component Loader: Carga el componente en memoria a pedido del *Component Manager* y lo deja disponible para su ejecución.
- Component Manager: Gestiona los componentes en tiempo de ejecución, determina su carga, intercambio por otro componente con la misma interface, y su descarga de memoria, de acuerdo a las necesidades del sistema.
- Component Repository: Todos los componentes del sistema están registrados en el Repositorio, de manera tal que el *Component Loader* puede ponerlos en ejecución a pedido del *Component Manager*. También lleva un conteo de referencias al

componente, a fin de determinar cuando el mismo ya no es necesario y puede ser descargado de memoria.

- Management Interface: Es la interface entre el componente y el *Component Framework*, normalmente se utiliza para informar el nombre y revisión del componente y la lista de otros componentes de los que depende, a fin de que puedan ser cargados por *Component Manager.*

Consecuencias:

Los sistemas con una arquitectura basada en componentes son muy estables, ya que la arquitectura básica del sistema se mantiene a medida que el mismo evoluciona y los defectos tienden a estar aislados dentro de un componente individual, por lo que su corrección solo tiene efectos locales y no suele propagarse al resto del sistema. Puede haber componentes con la misma funcionalidad provistos por diferentes vendedores y ser intercambiables (p.ej. librerías graficas para interface de usuario). Debido a que la interface es opaca, no es posible aprovechar optimizaciones dependientes de la implementación del componente. Por otra parte, el componente se utiliza como un todo, por lo que los recursos de la plataforma (tiempo de CPU, memoria, etc.) que deben utilizarse son todos los requeridos por el componente, aunque solo se utilice una pequeña parte de su funcionalidad.

Características Particulares:

El concepto fundamental de los componentes es su reemplazabilidad, por lo que su interface y características deben estar muy bien documentadas, no solo la forma de acceder a sus servicios, sino también las características no funcionales tales como la performance y el tamaño ocupado en memoria.

3.2. Concurrencia

3.2.1. Cola de Mensajes

Fuente:

Real Time Design Patterns: Robust Escalable Architecture for Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:

Concurrencia

Problema:

En sistemas con varios hilos de ejecución (tareas concurrentes) es necesario que estas intercambien información entre sí de tal manera que las tareas estén sincronizadas durante el intercambio y la información se intercambie de tal manera que no haya posibilidades de corrupción o carreras [*]

[*] Una carrera es una situación en la que el resultado del cómputo depende del orden de ejecución de los hilos, pero este orden no puede ser predicho.

Solución:

Una Cola de Mensajes es un mecanismo de comunicación asincrónico entre tareas o hilos de ejecución implementada mediante mensajes FIFO (First In – First Out) que permiten sincronizar e intercambiar información entre las tareas. Es una de las formas más simples de comunicación entre procesos y no presenta problemas de corrupción de la información porque la misma no se comparte por referencia, sino por valor. La tarea recibe una copia de la información y puede modificarla sin interferir con la tarea que envió la información. El diagrama de este patrón se muestra en la Figura 6

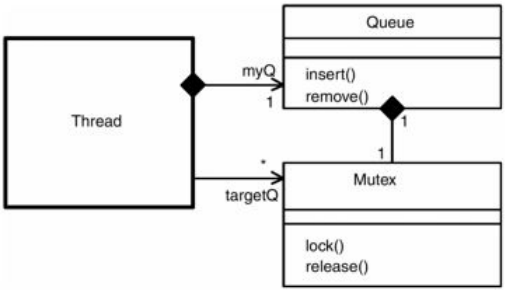

Fig. 6: Cola de Mensajes

Los elementos de este patrón son:

- Thread: este objeto es la tarea o hilo de ejecución. Puede enviar mensajes a otros objetos *Thread* y recibirlos y procesarlos cuando está en ejecución.
- Queue: es un contenedor de mensajes. El *Queue* guarda los mensajes hasta un *Thread* lo accede. El *Queue* se ejecuta en el contexto del *Thread* que lo invoque, por lo que es un recurso compartido, que debe ser protegido mediante el *Mutex*. Las operaciones mínimas que debe proveer son *insert()* y *remove()* para agregar y sacar mensajes.

- Mutex: Es un elemento de sincronismo, provee una operación de bloqueo *lock()* y una operación de desbloqueo *release()* para evitar accesos simultáneos desde dos *Threads* distintos. Cuando un *Thread* accede al *Queue*, si el *Mutex* está desbloqueado, pasa a estar bloqueado, impidiendo el acceso por parte de otros *Threads*. Cuando el *Thread* que accedió al *Queue* finaliza su acceso, el *Mutex* pasa al estado desbloqueado y otros *Threads* pueden acceder al *Queue*.

Consecuencias:
Como la información se pasa por valor, esto limita la complejidad de la colaboración entre tareas, y el proceso de copia de la información puede generar una sobrecarga a los recursos del sistema (memoria y/o tiempo de CPU).

Características Particulares:
En casi todo entorno multitarea se implementa alguna forma de Queue.

3.2.2.Interrupción

Fuente:
Real Time Design Patterns: Robust Escalable Architecture for Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:
Concurrencia

Problema:
En muchas aplicaciones, algunos eventos deben ser atendidos en forma rápida y eficientemente (sin consumir muchos recursos del sistema) y esto debe hacerse independientemente de lo que esté haciendo el sistema en ese momento, por lo que se dice que estos eventos generan una “Interrupción”. Cuando la respuesta que se debe dar al evento es simple, se aplica este patrón.

Solución:
El sistema operativo provee una Tabla de vectores de interrupción, que es un arreglo de direcciones de memoria, y hay un Gestor de Interrupciones, que gestiona las

direcciones de esta Tabla y las vincula con las funciones que procesaran la interrupción cuando ocurra. El diagrama de este Patrón se ve en la Figura 7:

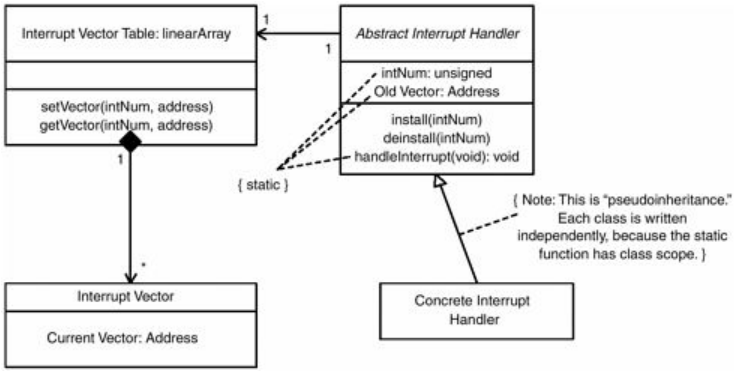

Fig. 7: Interrupción

Los elementos de este Patrón son:

- Abstract Interrupt Handler: Este objeto está vinculado con la Tabla de Vectores de Interrupción (*Interrupt Vector Table*) y provee una operación virtual *"handleInterrupt(void)"* que es la que procesa el evento. El método que implementa esta operación es provisto por la subclase *"Concrete Interrupt Handler".*
- Concrete Interrupt Handler: Es una subclase del *Abstract Interrupt Handler*, tiene las funciones necesarias para vincularse y desvincularse con la Tabla de Vectores de Interrupción (heredadas) e implementa el método *handleInterrupt()* para gestionar la interrupción. Si es necesario, puede encadenar con otro *Concrete Interrupt Handler* para que ambos objetos tengan la oportunidad de procesar la interrupción.
- Interrupt Vector: Es un tipo de dato abstracto que representa punteros a función.
- Interrupt Vector Table: Es un arreglo (array) de elementos *"Interrupt Vector"* asociados con distintos eventos. Cuando ocurre un evento cargado en este arreglo, se ejecuta el método correspondiente. Los distintos *Concrete Interrupt Handler* cargan aquí la dirección de su método *handleInterrupt().*

Consecuencias:

Cuando la respuesta al evento es simple y de corta duración, este patrón provee una forma eficiente de gestionar dicha respuesta. También se aplica cuando, si bien la respuesta puede ser compleja y/o de larga duración, la misma puede particionarse en una parte rápida (gestionada a través de handleInterrupt()), y una parte lenta que se puede procesar como parte de las tareas del sistema.

Si el procesamiento en el método handleInterrupt() es lento, se corre el riesgo de no poder atender otras interrupciones y/o que el sistema no pueda procesar otras tareas, ya que está permanentemente procesando interrupciones.
Por otra parte, es muy difícil compartir y/o transferir información entre los distintos handleInterrupt(), por la sobrecarga que ello implica al sistema.

Características Particulares:
Debe prevenirse la ocurrencia de una segunda interrupción durante el procesamiento de una primera interrupción, es decir, durante la ejecución del método handleInterrup(); a fin de evitar el uso del stack del procesador.
Los métodos handleInterrupt() no tienen parámetros de entrada o salida y son responsables de salvar el contexto del CPU antes de ejecutarse y de reestablecerlo al finalizar su ejecución.

3.2.3.Llamada protegida

Fuente:
Real Time Design Patterns: Robust Escalable Architecture for Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:
Concurrencia

Problema:
Muchas veces los esquemas de comunicación asincrónica no proveen una respuesta rápida para la comunicación entre objetos que están ejecutándose en distintos hilos de ejecución (Threads).

Solución:
Cuando se necesita una comunicación rápida entre objetos en diferentes Threads, una solución es simplemente ejecutar los métodos del objeto que se encuentra en otro Thread. Como no se sabe el estado del objeto, se debe emplear algún método de exclusión durante la llamada al método, por ejemplo un Mutex. Si el objeto en el otro

Thread quiere invocar alguno de sus propios métodos, el Mutex evita su ejecución hasta la finalización de la llamada. El diagrama de este Patrón se ve en la Figura 8:

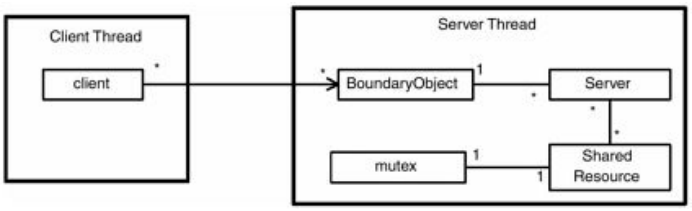

Fig. 8: Llamada Protegida

Los elementos de este Patrón son:

- Server Thread: Es el hilo de ejecución donde está el objeto remoto.
- Client Thread: Es el hilo de ejecución donde está el objeto Cliente que requiere invocar los métodos del objeto remoto
- Boundary Object: Provee una interface a los objetos que se ejecutan en el *Server Thread*
- Mutex: Es el elemento de exclusión que impide que se acceda al objeto en el *Server Thread* mientras dura la llamada desde el *Client Thread*
- Server: Administra el objeto *Shared Resource* y provee servicios al objeto Cliente a través del *Boundary Object*.
- Shared Resource: es el objeto en el *Server Thread* del cual se requiere que sus métodos puedan ser accedidos desde un objeto en el *Client Thread*

Consecuencias:

Este patrón permite el acceso a objetos que se encuentran en otro hilo de ejecución sin que se produzcan conflictos o riesgos de corrupción de datos.

Características Particulares:

Si el Server Object interactúa con un solo objeto Shared Resource, puede simplificarse este patrón de manera que el Boundary Object interactúa directamente con el Shared Resource, y el Boundary Object implementa el patrón clásico Facade [7]

3.2.4.Reunión

Real Time Design Patterns: Robust Escalable Architecture for Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:

Concurrencia

Problema:

En algunas situaciones es necesario que para la sincronización de distintos Threads, cada uno de ellos previamente haya cumplido una parte de cierta precondición; es decir, la sincronización se da cuando cada Thread cumplió su parte de la precondición.

Solución:

A medida que cada Thread cumple su parte de la precondición, la misma se registra en un objeto Reunión y se bloquea hasta que el objeto Reunión le indica que puede continuar su ejecución. Cuando todos los Threads que requieren la sincronización se registraron en el objeto Reunión, esto indica que se cumplió la totalidad de la precondición de sincronismo, por lo que el objeto Reunión desbloquea cada Thread registrado. Esto permite implementar precondiciones de complejidad arbitraria. El diagrama de este Patrón se ve en la Figura 9:

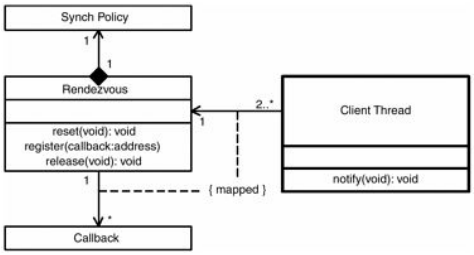

Fig. 9: Reunión

Los elementos de este patrón son:

- Callback: Es el objeto que permite notificar al *Client Thread* que se cumplió la condición de sincronismo.
- Client Thread: Son los *Thread* que requieren sincronismo, a medida que cada uno cumple su parte de la condición, se registra en el objeto *Rendezvous* y le pasa su *Callback*. El método *notify()* es invocado por el objeto *Rendezvous* para indicarle al *Client Thread* que se cumplió la condición de sincronismo y puede continuar su ejecución.
- Rendezvous: Es el objeto que gestiona la sincronización entre *Client Threads*; tiene un método *register(callback)* que es invocado por cada *Client Thread* para indicar que cumplió con su parte de la condición de sincronización.

- Synch Policy: Este objeto engloba la lógica de la precondición; en su forma más básica lleva la cuenta de cuantos *Client Thread* se registraron, y cuando se alcanza el número necesario, se cumplió la precondición. Si la precondición es más compleja, este objeto es el encargado de verificar si se cumplió, a fin de notificar a los *Client Thread*.

Consecuencias:
Este patrón permite implementar precondiciones complejas para el sincronismo entre Threads.

Características Particulares:
El objeto Synch Policy puede ser una maquina de estados, cuyas transiciones están determinadas por los Client Threads que se van registrando.

3.3. Gestión de la Memoria

3.3.1.Asignación Estática

Fuente:
Real Time Design Patterns: Robust Escalable Architecture for Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:
Gestión de la memoria

Problema:
Cuando se asigna memoria para las tareas, hay dos problemas asociados: 1) El tiempo de la asignación no es determinístico, ya que depende de la cantidad de memoria solicitada y 2) La fragmentación, que hace que si bien puede haber la cantidad de memoria solicitada disponible, la misma no está en un bloque continuo.

Solución:
En sistemas para cuyas tareas se puede estimar la memoria requerida y la misma se encuentra disponible en el hardware, dicha memoria es asignada durante la

inicialización del sistema y no es liberada durante la ejecución. De esta manera cuando cada tarea empieza a ejecutarse, ya tiene la memoria requerida asignada y no hay problemas de fragmentación. En un sistema orientado a objetos, todos los objetos serian creados durante la inicialización, y la estructura del patrón sería la de la Figura 10.

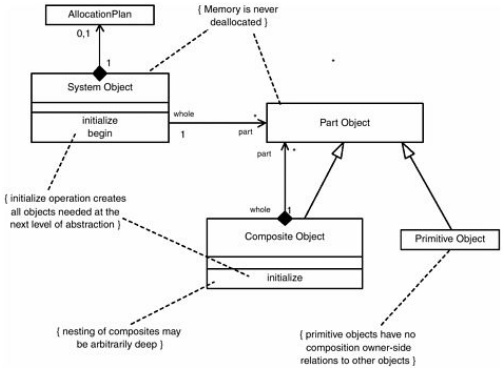

Fig. 10: Asignación Estática

Los elementos de este patrón son:

- Allocation Plan: Es donde se establece el orden en que se instancian los *Composite Objects*.
- Composite Object: son objetos que tienen relación de composición con otros objetos, que pueden ser también *composite objects o primitive objects*. Son responsables de la creación de todos los objetos con los que tienen relación de composición.
- Part Object: Es una clase abstracta ancestro de los objetos *Composite Object* y *Primitive object*. De esta manera el *system object* puede inicializar la cadena de *composite objects*.
- Primitive object: son objetos que no crean otros objetos. Todos los *primitive objects* son creados por los *composite objects.*
- System object: contiene el *allocation plan* y en base a este va instanciando los *composite objects*. Una vez creados todos los objetos, invoca el método *begin()* para comenzar la ejecución del sistema

Consecuencias:

Este patrón es aplicable cuando se conoce la cantidad de memoria que requiere el sistema y la misma está disponible en el hardware. El tiempo de ejecución de las tareas

es más predecible, ya que no está la incertidumbre asociada al tiempo de asignación de memoria.

Características Particulares:
Los sistemas que implementan este patrón, comparados con los que implementan otros patrones de gestión de memoria, suelen tener un tiempo de arranque mayor, pero luego de esta etapa, el sistema suele ejecutarse más rápido; como contrapartida, aplicar este patrón suelen requerir más memoria en el hardware.

3.3.2.Asignación desde una Reserva

Fuente:
Real Time Design Patterns: Robust Escalable Architecture for Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:
Gestión de la memoria

Problema:
Hay situación en las que no es posible conocer la cantidad de memoria total que necesitará el sistema durante su ejecución y/o la misma no está disponible en el hardware, por otra parte se necesitan distintos conjuntos de objetos en diferentes momentos durante la ejecución.

Solución:
Durante la inicialización se crea una cierta cantidad (reserva) de objetos, que están disponibles para ser usados por distintos clientes (otros objetos, con quienes tienen una relación de composición). Cuando un cliente lo requiere, se le asignan los objetos de la reserva, y cuando el cliente no los necesita, los devuelve a la reserva, quedando los mismos disponibles para ser usados más adelante. El diagrama de este patrón se ve en la Figura.11

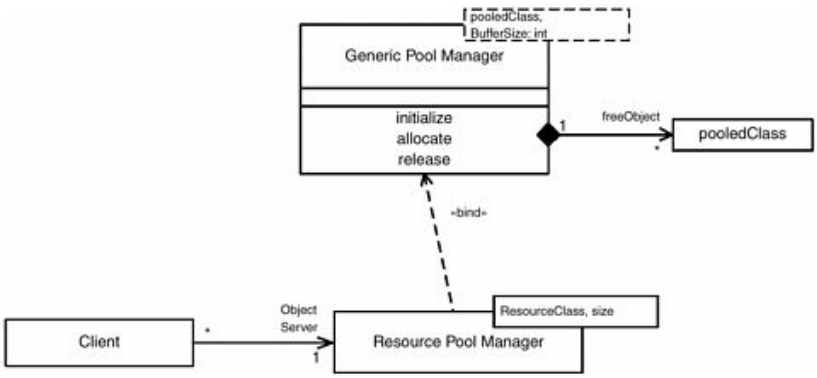

Fig. 11: Asignación desde una Reserva

Los elementos de este patrón son:

- Client: Es un objeto del sistema que necesita uno o más objetos del tipo *resource class*, para ello invoca a *resourcePool::allocate()* para tomar los objetos de la reserva y los devuelve mediante *resourcePool::release()*.
- Generic Pool Manager: Es una clase parametrizable (un *template*) que usa los parámetros *pooledClass* y *BuffeSize* para especificar la clase de objetos que reserva y el número total de ellos en la reserva.
- PooledClass: es el parámetro pasado a *Generic Pool Manager* para crear una reserva de objetos específicos.
- Resource Pool Manager: es una instancia parametrizada de *Generic Pool Manager* en la que una clase específica (*ResourceClass*) y una cantidad de objetos (*size*) fueron pasados como parámetro. Puede haber varios *Resource Pool Manager* en el sistema, pero hay uno solo por cada *ResourceClass*.

Consecuencias:

Al igual que en la asignación estática, la memoria es asignada en el inicio y no es liberada, por lo que no hay incertidumbre en la ejecución de las tareas por el tiempo de asignación; ni fragmentación de la memoria; pero el sistema gana la capacidad de gestionar la asignación no determinística de determinados objetos.

Este patrón es especialmente aplicable para sistemas en los que un cierto número de instancias de un objeto son necesitadas por distintos clientes (p.ej. objetos para representar mensaje entre tareas). De esta manera, las instancias son asignadas a los clientes de acuerdo al criterio de “según necesidad” y pueden ser reutilizados.

Características Particulares:

Es una variación del patrón “Abstract Factory Method”, con la diferencia de que en vez de crearse los objetos a medida que son requeridos, los mismos son asignados de la reserva.

3.3.3.Zona de memoria de tamaño fijo

Fuente:

Real Time Design Patterns: Robust Escalable Architecture for Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:

Gestión de la memoria

Problema:

La gestión de memoria de algunos sistemas es tan compleja e impredecible que es necesario poder asignarla en forma dinámica; pero por distintas razones no se puede reubicar la memoria ya asignada a fin de mitigar la fragmentación.

Solución:

Fijando un tamaño de bloque de memoria constante por asignación, de tal manera que cubra el peor caso de asignación de memoria (el mayor requerimiento), se evita la fragmentación, ya que este tamaño de bloque siempre cubrirá los requerimientos de asignación de memoria. El diagrama de este patrón se puede ver en la Figura. 12

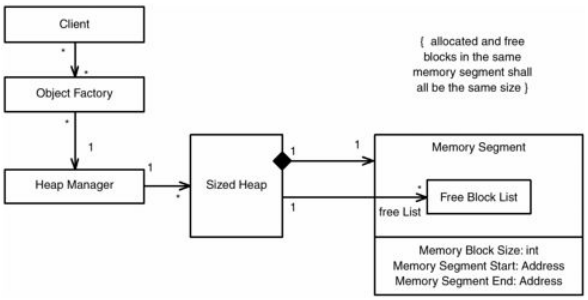

Fig. 12: Zona de Memoria de Tamaño Fijo

Los elementos de este patrón son los siguientes:

- Client: Es el usuario de los objetos cuya memoria es asignada en bloques.

- Free Block List: Es una lista de los bloques de memoria disponibles dentro de un *Memory Segment*.
- Heap Manager: Gestiona la asignación de la memoria, tiene uno o varios objetos *Sized Heap*, a quienes les solicita los bloques de memoria según la necesidad.
- Memory Segment: Es una zona de memoria dividida en bloques de igual tamaño que pueden ser asignados o liberados; tiene un objeto *Free Block List*, que lleva la cuenta de los bloques libres. Posee atributos que indican el tamaño de bloque que asigna (*Memory Block Size*), y las direcciones de inicio y finalización de la zona de memoria (*Memory Segment Start* y *Memory Segment End*)
- Object Factory: Es el responsable de crear y destruir los objetos solicitados por los *Client*, utilizando memoria provista por el *Heap Manager* y devolviéndosela cuando ya no es necesaria.
- Sized Heap: Este objeto gestiona la asignación y liberación de bloques de un *Memory Segment*. Devuelve una referencia al bloque de memoria asignado, y cuando recibe de regreso dicha referencia (cuando la memoria es liberada), la coloca en la *Free Block List* para que el bloque pueda ser reutilizado.

Consecuencias:

Si bien este patrón gestiona la memoria en forma dinámica y evita la fragmentación, hace un uso ineficiente de la misma, ya que siempre se asigna la mayor cantidad de memoria que se podría llegar a necesitar en una asignación, pero si se necesita una cantidad menor, el resto del bloque de memoria esta desperdiciado. Por otra parte si bien esta minimizado, el problema del indeterminismo en el tiempo de asignación de la memoria está presente. El uso de este patrón no evita la pérdida de memoria (memory leak), ya que cada *Client* es responsable de notificar al *Object Factory* que el objeto requerido ya no es necesario.

Características Particulares:

Una variante de este patrón establece distintos tamaños de bloque asignable (mediante varios objetos *Memory Segment*). En general se implementan los 2 o 3 tamaños que serían más requeridos, así como también el mayor bloque necesario. De esta manera se logra una mayor granularidad en la asignación de memoria y se reduce el

desperdicio, al poder asignarse un bloque de memoria más cercano a la memoria necesaria.

3.3.4.Puntero Inteligente

Fuente:

Real Time Design Patterns: Robust Escalable Architecture for Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:

Gestión de la memoria

Problema:

El uso de punteros a direcciones de memoria tiene asociados algunos riesgos: un puntero es destruido, pero la memoria a la que hace referencia no es liberada (memory leak); la memoria es liberada pero el puntero se sigue usando para acceder a ella (dangling pointer), se accede a memoria a través del puntero sin antes haberlo inicializado (unitialized pointer), se accede a datos utilizando el puntero como base y sumando un incremento en forma incorrecta (bad pointer arithmetic), etc.

Solución:

Los problemas antes mencionados se deben a que el puntero no es un objeto en sí mismo, sino un dato (una referencia a una dirección de memoria); por lo que no hay un chequeo de validez de las operaciones sobre el puntero. La solución consiste entonces en hacerlo un objeto, por lo que pasa a tener constructor y destructor, y operaciones sobre las que se puede hacer un chequeo de validez. Hay dos variantes, en una el elemento referenciado tiene información de acerca de su referenciación (basic variant); en la otra el elemento referenciado no tiene información acerca de su referenciación, por lo que mediante composición se crea un objeto que contiene el elemento referenciado y la información de referenciación (wrapper variant). El diagrama de este patrón se ve en la Figura.13

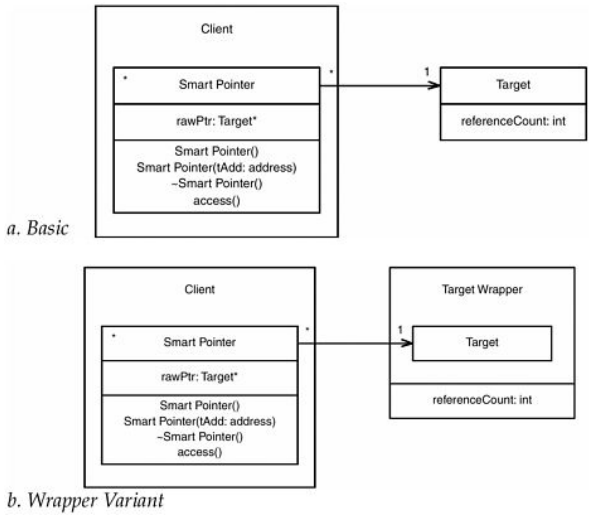

Fig. 13: Puntero Inteligente

Los elementos de este patrón son:

- Client: Es el objeto que tiene un *Smart Pointer* para usar como referencia.
- Smart Pointer: es un objeto que tiene como atributo el puntero concreto (*rawPointer*), constructores (*SmartPointer()* y *SmartPointer(tAdd: Address)*), destructor (*~SmartPointer()*), y operaciones de acceso (*access()*). Se utilizan dos constructores porque cuando se crea el *Target*, el *SmartPointer* asociado se crea con el constructor sin parámetros (el cual fija *referenceCount* en 1). Cuando es necesario crear más referencias al *Target*, se utiliza el constructor que recibe como parámetro la dirección del *Target*, el cual inicializa su atributo de puntero concreto (*rawPointer*) con el parámetro recibido, e incrementa *referenceCount*. El destructor decrementa *Target::referenceCount*, y si el decremento llega a 0, destruye *Target* y libera la memoria apuntada por *rawPointer*.
- Target: es la referencia a la que necesita acceder el *Client*. En la versión básica (*Basic*) tiene un atributo *referenceCount* que lleva la cuenta de cuantos *SmartPointers* lo referencian.
- Target Wrapper: En la segunda variante de este patrón, como el *Target* no tiene un conteo de cuantos *SmartPointers* lo referencian (información necesaria para saber cuándo destruir el *Target* y liberar la memoria), es necesario construir este objeto, el cual mediante composición integra el *Target* y el contador de referencias *referenceCount*.

Consecuencias:

Con este patrón es posible evitar los problemas asociados a punteros, ya que es posible implementar reglas de validez sobre el uso del puntero y el acceso a su referencia.

Características Particulares:

Aun aplicando este patrón es posible que haya memory leaks, esto sucede si hay una referencia cruzada entre dos objetos (ambos se apuntan entre sí). Como se ve en la Figura 14

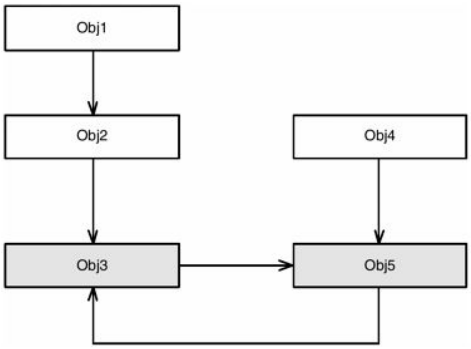

Fig. 14: Referencias Cruzadas entre Objetos

Al destruirse los objetos Obj2 y Obj4, los referenceCounter de Obj3 y Obj5 se decrementan a 1 (ya que ambos se apuntan entre sí formando un ciclo), pero ninguno de los dos es accesible desde el resto de la aplicación (p.ej. desde Obj1). La única solución a esto es verificar en la etapa de diseño que no se produzcan referencias cíclicas entre los objetos; si las mismas son inevitables, no es recomendable utilizar este patrón en esos objetos.

3.3.5.Recolector de basura

Fuente:

Real Time Design Patterns: Robust Escalable Architecture for Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:

Gestión de la memoria

Problema:

La asignación de memoria dinámica mediante punteros tiene asociados algunos riesgos: un puntero es destruido, pero la memoria a la que hace referencia no es liberada (memory leak); o la memoria es liberada pero el puntero se sigue usando para acceder a ella (dangling pointer). Esto ocurre porque es responsabilidad del programador de la aplicación o tarea el liberar la memoria utilizada.

Solución:

Al aplicar este patrón, deja de ser responsabilidad del programador la liberación de memoria de sus tareas, sino que dicha liberación pasa a ser responsabilidad de una tarea del sistema. Este patrón trabaja en dos etapas, denominadas “marcado” y liberación” (*marking* and *reclamation*) La etapa de *marking* es cuando los objetos son creados, en ese momento se marcan como “objetos vivos”. La etapa *reclamation* sucede cuando el sistema detecta poca memoria libre disponible, o mediante una invocación explicita (por ejemplo en forma periódica). En esta etapa cada objeto raíz (*root object*) es analizado para encontrar todos sus objetos vinculados (mediante relaciones de composición). Todos los objetos que no puedan ser encontrados a partir de un root object o sus objetos vinculados se marcan como “dead objects”; finalizado el proceso de análisis, todos los objetos marcados como dead objects son eliminados y su memoria liberada. Un diagrama de este patrón se ve en la Figura.15

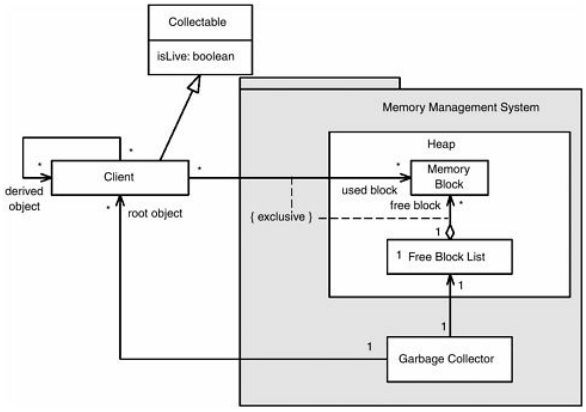

Fig. 15: Recolector de Basura

Los elementos de este patrón son los siguientes:

- Client: Es un objeto que necesita reservar memoria, es una subclase de colectable y contiene punteros a *derived objects*, permitiendo al *garbage collector* buscar

los objetos vinculados. Cuando se crean, los objetos se marcan como vivos mediante su atributo *isLive*.

- Collectable: es la clase base para los *client*, proveyendo el atributo *isLive* que utilizar el *garbage collector*.
- Free block list: lista de bloques de memoria disponibles.
- Garbage collector: Gestiona la liberación de memoria, lleva una lista de todos los objetos creados y está vinculado a los *root objects*. Durante la etapa de *reclamation*, todos los objetos de la lista se marcan como *dead objects* (poniendo su atributo *isLive* en falso), luego se parte de los *root objects* y se los marca como vivos (poniendo su atributo *isLive* en verdadero). Lo mismo se hace con los objetos asociados a cada *root object* (los *derived objects*). Finalmente los objetos de la lista cuyo atributo *isLive* continua en falso son eliminados y su memoria reclamada.
- Heap: Es el objeto propietario de los *Memory Blocks* y la lista *Free block list*.
- Memory Block: es un bloque de memoria física, puede estar libre, en cuyo caso esta referenciado por la *Free Block list*, pero no por un *client*, o puede estar reservado, por lo que estará referenciado por un *client*, pero no por la *Free Block list*, de ahí la marcación de "*exclusive*" que se ve en el diagrama.

Consecuencias:

Aplicar este patrón elimina los memory leaks y los dangling points, pero genera una tarea más, que al ejecutarse debe suspender la actividad normal del sistema hasta su finalización, lo que implica aumentar la carga del sistema (overead); y se pierde predictibilidad en la ejecución de tareas, ya que en la etapa de diseño no se sabe en qué momento será necesario liberar memoria asignada y/o cuanto tiempo llevara esta tarea.

Características Particulares:

Si bien este patrón previene la aparición de memory leaks y dangling pointers, no evita la fragmentación de la memoria. Para ello hay una variante denominada "Garbage Compactor"; en el que se mantienen dos copias de cada *memory segment*. Durante la etapa de *reclamation*, los objetos *root object* y sus objetos asociados (*derived objects*) son copiados de un *memory segment* al otro en forma contigua cuando se fija su

atributo *isLive* en verdadero; de esta manera al finalizar la etapa de *reclamation* la memoria asignada y la memoria libre forman ambas sendos bloques contiguos en el *memory segment* sobre el que se hizo la copia. El inconveniente con esta variante es que genera un overhead aun mayor que el Garbage Collector por la copia de cada objeto de un *memory segment* al otro, y que cada *memory segment* ocupa el doble de memoria física (ya que debe haber dos copias)

Capitulo 4. Patrones para S.E. con requerimientos de tiempo real

4.1. Especificación de S.E. con req. de tiempo real

4.1.1.Ausencia

Fuente:

Patterns in Property Specifications for Finite State Verification, Matthew Dwyyer, George Avrunin, James Corbett, ICSE'99, mayo de 1999

Área de Aplicación:

Tiempo Real – Sistemas definidos como Maquinas de Estado

Problema:

Este patrón se utiliza para caracterizar la propiedad de un sistema que un determinado estado/evento P no debe ocurrir nunca dentro del alcance.

Solución

Propiedad expresada en Castellano:

"(alcance) **Nunca** se da el caso que suceda P"

Propiedad expresada en Estructured English:

"(scope) It is **never** the case that P holds"

Consecuencias:

Utilizar una gramática estructurada en la especificación de requerimientos permite aplicar herramientas de verificación sobre los mismos.

Características Particulares:

Patrón de especificación para la Especificación de un Sistema como un conjunto de Estados + Transiciones asociadas a Eventos.

4.1.2.Universalidad

Fuente:

Patterns in Property Specifications for Finite State Verification, Matthew Dwyyer, George Avrunin, James Corbett, ICSE'99, mayo de 1999

Área de Aplicación:

Tiempo Real – Sistemas definidos como Maquinas de Estado

Problema:

Este patrón se utiliza para caracterizar la propiedad de un sistema que un determinado estado/evento P sucede durante todo el alcance.

Solución

Propiedad expresada en castellano:

"(alcance) P sucede **siempre**"

Propiedad expresada en Estructured English:

"(scope) It is **always** the case that P holds"

Consecuencias:

Utilizar una gramática estructurada en la especificación de requerimientos permite aplicar herramientas de verificación sobre los mismos.

Características Particulares:

Patrón de especificación para la Especificación de un Sistema como un conjunto de Estados + Transiciones asociadas a Eventos.

4.1.3.Existencia

Fuente:

Patterns in Property Specifications for Finite State Verification, Matthew Dwyyer, George Avrunin, James Corbett, ICSE'99, mayo de 1999

Área de Aplicación:

Tiempo Real – Sistemas definidos como Maquinas de Estado

Problema:
Este patrón se utiliza para caracterizar la propiedad de un sistema que un determinado estado/evento sucede siempre en algún momento dentro del alcance.

Solución
Propiedad expresada en castellano:
"(alcance) P **en algún momento** sucede"
Propiedad expresada en Estructured English:
"(scope) P **eventually** holds"

Consecuencias:
Utilizar una gramática estructurada en la especificación de requerimientos permite aplicar herramientas de verificación sobre los mismos.

Características Particulares:
Patrón de especificación para la Especificación de un Sistema como un conjunto de Estados + Transiciones asociadas a Eventos.

4.1.4.Existencia Acotada

Fuente:
Patterns in Property Specifications for Finite State Verification, Matthew Dwyyer, George Avrunin, James Corbett, ICSE'99, mayo de 1999

Área de Aplicación:
Tiempo Real – Sistemas definidos como Maquinas de Estado

Problema:
Este patrón se utiliza para caracterizar la propiedad de un sistema que un determinado estado/evento P sucede k veces dentro del alcance.

Solución
Propiedad expresada en castellano:

"(alcance) **suceden k** transiciones a los estados en los cuales ocurre P"
Propiedad expresada en Estructured English:
"(scope) Transitions to states in which P holds **occur k times**"

Consecuencias:
Utilizar una gramática estructurada en la especificación de requerimientos permite aplicar herramientas de verificación sobre los mismos.

Características Particulares:
Variantes:

a)El evento debe suceder como máximo k veces:
"(alcance) **suceden como máximo k** transiciones a los estados en los cuales ocurre P"
"(scope) Transitions to states in which P holds **occur at most k times"**
b)El evento debe suceder como mínimo k veces:
"(alcance) **suceden como mínimo k** transiciones a los estados en los cuales ocurre P"
"(scope) Transitions to states in which P holds **occur at least k times"**

Patrón de especificación para la Especificación de un Sistema como un conjunto de Estados + Transiciones asociadas a Eventos.

4.1.5.Precedencia

Fuente:
Patterns in Property Specifications for Finite State Verification, Matthew Dwyyer, George Avrunin, James Corbett, ICSE'99, mayo de 1999

Área de Aplicación:
Tiempo Real – Sistemas definidos como Maquinas de Estado

Problema:
Este patrón se utiliza para caracterizar la propiedad de un sistema que un determinado estado/evento P debe ser precedido por un estado/evento Q dentro del alcance.

Solución

Propiedad expresada en castellano:

"(alcance) Siempre se cumple que si sucede P, entonces **previamente sucedió** Q"

Propiedad expresada en Estructured English:

"(scope) It is always the case that if P holds, then Q **previously held**"

Consecuencias:

Utilizar una gramática estructurada en la especificación de requerimientos permite aplicar herramientas de verificación sobre los mismos.

Características Particulares:

Patrón de especificación para la Especificación de un Sistema como un conjunto de Estados + Transiciones asociadas a Eventos.

4.1.6.Cadena de Precedencia

Fuente:

Patterns in Property Specifications for Finite State Verification, Matthew Dwyyer, George Avrunin, James Corbett, ICSE'99, mayo de 1999

Área de Aplicación:

Tiempo Real – Sistemas definidos como Maquinas de Estado

Problema:

Este patrón se utiliza para caracterizar la propiedad de un sistema que un determinado estado/evento P debe ser precedido por una secuencia de estados/eventos Q1…Qn dentro del alcance.

Solución

Propiedad expresada en castellano:

"(alcance) Siempre se cumple que si sucede P, **y luego sucede Qn, entonces previamente sucedieron Q1…Qn-1**"

Propiedad expresada en Estructured English:

"(scope) It is always the case that if P holds, **and is succeeded by Qn, then Q1…Qn-1 previously held**"

Consecuencias:

Utilizar una gramática estructurada en la especificación de requerimientos permite aplicar herramientas de verificación sobre los mismos.

Características Particulares:

Este patrón representa la secuencia Q1…Qn-1 -> P -> Qn

Variantes:

a)Esta variante representa la secuencia Q1…Qn-1 -> Qn -> P :

"(alcance) Siempre se cumple que si sucede P, **entonces Qn sucedió previamente y fue precedido por Q1…Qn-1**"

"(scope) It is always the case that if P holds, **then Qn previously held and was preceded by Q1…Qn-1**".

Es una generalización del patrón *Precedencia.*

Patrón de especificación para la Especificación de un Sistema como un conjunto de Estados + Transiciones asociadas a Eventos.

4.1.7.Respuesta

Fuente:

Patterns in Property Specifications for Finite State Verification, Matthew Dwyyer, George Avrunin, James Corbett, ICSE'99, mayo de 1999

Área de Aplicación:

Tiempo Real – Sistemas definidos como Maquinas de Estado

Problema:

Este patrón se utiliza para caracterizar la propiedad de un sistema que un determinado estado/evento P es seguido por un estado/evento Q en algún momento dentro del alcance.

Solución

Propiedad expresada en castellano:

"(alcance) Siempre se cumple que si sucede P, entonces **en algún momento sucede** Q"

Propiedad expresada en Estructured English:

"(scope) It is always the case that if P holds, then Q **eventually holds**"

Consecuencias:

Utilizar una gramática estructurada en la especificación de requerimientos permite aplicar herramientas de verificación sobre los mismos.

Características Particulares:

Patrón de especificación para la Especificación de un Sistema como un conjunto de Estados + Transiciones asociadas a Eventos.

4.1.8.Cadena de Respuesta

Fuente:

Patterns in Property Specifications for Finite State Verification, Matthew Dwyyer, George Avrunin, James Corbett, ICSE'99, mayo de 1999

Área de Aplicación:

Tiempo Real – Sistemas definidos como Maquinas de Estado

Problema:

Este patrón se utiliza para caracterizar la propiedad de un sistema que un determinado estado/evento P debe ser seguido por una secuencia de estados/eventos Q1…Qn en algún momento dentro del alcance.

Solución

Propiedad expresada en castellano:

"(alcance) Siempre se cumple que si sucede P, entonces **en algún momento sucede Q1 y es seguido por Q2…Qn**"

Propiedad expresada en Estructured English:

"(scope) It is always the case that if P holds, **then Q1 eventually holds and is succeeded by Q2…Qn**"

Consecuencias:

Utilizar una gramática estructurada en la especificación de requerimientos permite aplicar herramientas de verificación sobre los mismos.

Características Particulares:

Este patrón representa la secuencia P -> Q1…Qn

Variantes:

a)Esta variante representa la secuencia (P -> Qn) -> Q1…Qn-1

"(alcance) Siempre se cumple que si sucede P y es sucedido por Qn, entonces **en algún momento suceden Q1…Qn-1**"

"(scope) It is always the case that if P holds, **and is succeeded by Qn, then Q1…Qn-1 eventually holds after Qn**".

Es una generalización del patrón *Respuesta.*

Patrón de especificación para la Especificación de un Sistema como un conjunto de Estados + Transiciones asociadas a Eventos.

4.1.9.Cadena Restringida

Fuente:

Patterns in Property Specifications for Finite State Verification, Matthew Dwyyer, George Avrunin, James Corbett, ICSE'99, mayo de 1999

Área de Aplicación:

Tiempo Real – Sistemas definidos como Maquinas de Estado

Problema:

Este patrón se utiliza para caracterizar la propiedad de un sistema que un determinado estado/evento P en algún momento es seguido por una secuencia de estados/eventos Q1…Qn dentro del alcance, pero el evento R no debe suceder entre los eventos Q1 y Qn.

Solución

Propiedad expresada en castellano:

"(alcance) Siempre se cumple que si sucede P, **entonces en algún momento sucede Q1 y es sucedido por Q2…Qn, pero R no sucede entre Q1 y Qn**"

Propiedad expresada en Estructured English:

"(scope) It is always the case that if P holds, **then Q1 eventually holds and is succeeded by Q2…Qn, where R does not hold between Q1 and Qn**"

Consecuencias:

Utilizar una gramática estructurada en la especificación de requerimientos permite aplicar herramientas de verificación sobre los mismos.

Características Particulares:

Este patrón representa la secuencia P -> Q1…Qn, con la particularidad que R no debe suceder entre Q1 y Qn.

Patrón de especificación para la Especificación de un Sistema como un conjunto de Estados + Transiciones asociadas a Eventos.

4.1.10.Duración Mínima

Fuente:

Real Time Specification Patterns, Sascha Konrad, Betty Cheng, ICSE05, Mayo de 2005

Área de Aplicación:

Tiempo Real – Sistemas definidos como Maquinas de Estado

Problema:

Este patrón se utiliza para caracterizar la propiedad de un sistema que cada vez que se cumple cierta condición P, esta condición se mantiene por lo menos durante cierta cantidad de tiempo

Solución
Propiedad expresada en castellano:
"(alcance) Siempre se cumple que, una vez que sucede P, se mantiene **por al menos C unidades de tiempo**"
Propiedad expresada en Estructured English:
"(scope) It is always the case that once P becomes satisfied, it holds for **at least C time units**"

Consecuencias:
Utilizar una gramática estructurada en la especificación de requerimientos permite aplicar herramientas de verificación sobre los mismos.

Características Particulares:
Patrón de especificación para la Especificación de un Sistema como un conjunto de Estados + Transiciones asociadas a Eventos

4.1.11.Duración Máxima

Fuente:
Real Time Specification Patterns, Sascha Konrad, Betty Cheng, ICSE05, Mayo de 2005

Área de Aplicación:
Tiempo Real– Sistemas definidos como Maquinas de Estado

Problema:
Este patrón se utiliza para caracterizar la propiedad de un sistema que cada vez que se cumple cierta condición P, esta condición se mantiene como máximo durante cierta cantidad de tiempo

Solución

Propiedad expresada en castellano:

"(alcance) Siempre se cumple que, una vez que sucede P, se mantiene **durante menos de C unidades de tiempo"**

Propiedad expresada en Estructured English:

"(scope) It is always the case that once P becomes satisfied, it holds for **less than C time units**"

Consecuencias:

Utilizar una gramática estructurada en la especificación de requerimientos permite aplicar herramientas de verificación sobre los mismos.

Características Particulares:

Patrón de especificación para la Especificación de un Sistema como un conjunto de Estados + Transiciones asociadas a Eventos

4.1.12.Recurrencia Acotada

Fuente:

Real Time Specification Patterns, Sascha Konrad, Betty Cheng, ICSE05, Mayo de 2005

Área de Aplicación:

Tiempo Real– Sistemas definidos como Maquinas de Estado

Problema:

Este patrón se utiliza para caracterizar la propiedad de un sistema que determinado evento P suceda con una periodicidad no menor a un cierto valor

Solución

Propiedad expresada en castellano:

"(alcance) Siempre se cumple que P sucede **por lo menos cada C unidades de tiempo**"

Propiedad expresada en Estructured English:

"(scope) It is always the case that P holds **at least every C time units**

Consecuencias:

Utilizar una gramática estructurada en la especificación de requerimientos permite aplicar herramientas de verificación sobre los mismos.

Características Particulares:

Patrón de especificación para la Especificación de un Sistema como un conjunto de Estados + Transiciones asociadas a Eventos.

4.1.13.Respuesta Acotada

Fuente:

Real Time Specification Patterns, Sascha Konrad, Betty Cheng, ICSE05, Mayo de 2005

Área de Aplicación:

Tiempo Real – Sistemas definidos como Maquinas de Estado

Problema:

Este patrón se utiliza para caracterizar la propiedad de un sistema que, dada la ocurrencia de cierto evento P, otro evento S (normalmente la respuesta del sistema) debe suceder dentro de un cierto tiempo límite y no excederlo

Solución

Propiedad expresada en castellano:

"(alcance) Siempre se cumple que si sucede P, entonces S sucede **después de cómo mucho C unidades de tiempo**"

Propiedad expresada en Estructured English:

"(scope) It is always the case that if P holds, then S Holds **after at most C time units**"

Consecuencias:

Utilizar una gramática estructurada en la especificación de requerimientos permite aplicar herramientas de verificación sobre los mismos.

Características Particulares:

Patrón de especificación para la Especificación de un Sistema como un conjunto de Estados + Transiciones asociadas a Eventos

4.1.14.Invarianza Acotada

Fuente:

Real Time Specification Patterns, Sascha Konrad, Betty Cheng, ICSE05, Mayo de 2005

Área de Aplicación:

Tiempo Real – Sistemas definidos como Maquinas de Estado

Problema:

Este patrón se utiliza para caracterizar la propiedad de un sistema que, dada la ocurrencia de cierto evento P, otro evento S (normalmente la respuesta del sistema) debe suceder y mantenerse durante por lo menos un cierto tiempo.

Solución

Propiedad expresada en castellano:

"(alcance) Siempre se cumple que, si sucede P, entonces S sucede **durante por lo menos C unidades de tiempo**"

Propiedad expresada en Estructured English:

"(scope) It is always the case that if P holds, then S Holds **for at least C time units**"

Consecuencias:

Utilizar una gramática estructurada en la especificación de requerimientos permite aplicar herramientas de verificación sobre los mismos.

Características Particulares:

Patrón de especificación para la Especificación de un Sistema como un conjunto de Estados + Transiciones asociadas a Eventos.

4.2. Ejecución de Tareas con requerimientos de tiempo real

4.2.1.Ejecución cíclica

Fuente:

Real Time Design Patterns: Robust Escalable Architecture For Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:

Sistemas multitarea – Estrategias para la programación de la tarea a ejecutar

Problema:

En algunos S.E. multitarea con requerimientos muy exigentes de memoria y poca capacidad de cómputo puede no ser viable utilizar un RTOS para coordinar la ejecución de las tareas, por lo que se necesita implementar alguna forma de multitarea que no requiera recursos del sistema.

Solución:

La forma más simple de multitarea es que, cuando una tarea finaliza su ejecución, el programador de tareas invoca a la siguiente tarea, y así sucesivamente. Al finalizar la última tarea, el programador de tareas invoca a la primera y el ciclo se repite. El diagrama de este patrón se ve en la Figura 16.

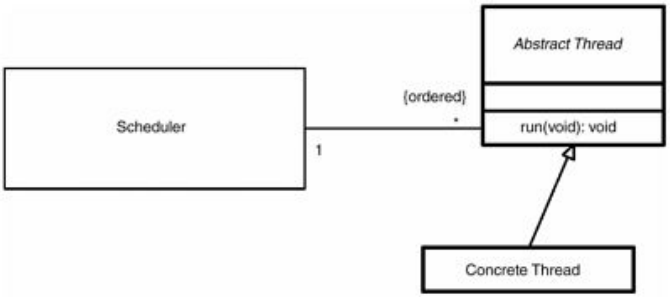

Fig. 16: Ejecución Cíclica

Los elementos de este patrón son:

- Abstract Thread: Es una clase abstracta que declara el método *run(),* el cual es usado en cada *Concrete Thread* para invocar su ejecución.

- Concrete Thread: Es el objeto que implementa la funcionalidad de la tarea. Es responsabilidad de cada tarea devolver el control al *Scheduler* al finalizar su ejecución.
- Scheduler: Es el objeto programador de tareas que inicializa el sistema, carga las tareas e invoca el método *run()* de cada tarea.

Consecuencias:

Este es un patrón de ejecución de tareas sumamente simple de implementar, pero no tiene flexibilidad. Como las tareas no pueden agregarse o quitarse, se aplica a sistemas con pocas tareas que se ejecutan iterativamente durante todo el tiempo de ejecución del sistema. El tiempo de ejecución de las tareas es totalmente predecible. No hay prioridad de ejecución de las tareas.

Características Particulares:

Para aplicar este patrón deben darse una serie de características:

- El número de tareas es constante durante toda la ejecución del sistema (no se agregan o sacan tareas)
- El tiempo de ejecución de cada tarea en cada ciclo no es importante o es constante.
- Las tareas son independientes entre sí.
- Se debe asegurar que los recursos que usa cada tarea son liberados al momento en que esta devuelve el control al Scheduler
- Existe un orden secuencial de ejecución que es adecuado para todas las situaciones previstas de uso del sistema.

4.2.2.Ejecución en Ronda

Fuente:

Real Time Design Patterns: Robust Escalable Architecture For Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:

Sistemas multitarea – Estrategias para la programación de la tarea a ejecutar

Problema:

En S.E. multitarea en algunos casos es más importante que las tareas avancen en su ejecución a que se cumpla cierto requerimiento de tiempo de ejecución.

Solución:

Este patrón implementa una política de asignación del procesador en la cual cada tarea recibe una porción igual de tiempo del procesador, asegurando su posibilidad de progresar en la ejecución, aun cuando no completen su actividad en el tiempo de procesador asignado. El diagrama de este patrón se ve en la Figura 17.

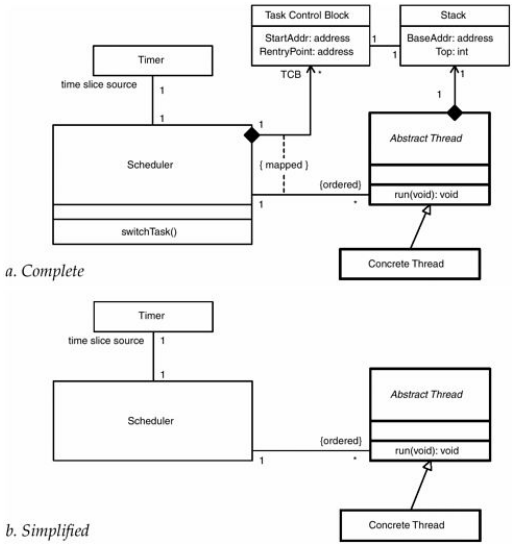

Fig. 17: Ejecución en Ronda

Los elementos de este patrón son:

- Abstrac Thread: Es una clase abstracta que declara el método *run(),* el cual es usado en cada *Concrete Thread* para invocar su ejecución.
- Concrete Thread: Es el objeto que implementa la funcionalidad de la tarea.
- Scheduler: Es el objeto programador de tareas que inicializa el sistema, carga las tareas e invoca el método *run()* de cada tarea. Las tareas pueden devolver el control voluntariamente al *Scheduler*, o ser interrumpidas por el *Scheduler* cuando este recibe una señal del objeto *Timer*.
- Stack: Cada objeto *Concrete Thread* tiene asociado un objeto *Stack* para guardar toda la información asociada a la tarea (su "contexto"), para que al resumir su ejecución, puedan continuar la ejecución en el punto en que se interrumpió.

- Task Control Block: El *Scheduler* tiene un objeto *Task Control Block* por cada tarea en ejecución, allí guarda información asociada a la tarea, como por ejemplo el punto de ejecución en el que fue interrumpida.
- Timer: Este objeto envía señales periódicas al *Scheduler* para indicar que se cumplió el tiempo de ejecución de la tarea actual. En general el *Timer* está basado en un contador en hardware. El *Scheduler* ejecuta su método *switchTask()* cada vez que recibe la señal periódica.

Consecuencias:

Cada tarea tiene un tiempo de ejecución parejo. Este patrón evita que una tarea bloqueada impida la ejecución de otras tareas, ya que el tiempo de procesador se divide entre todas las tareas.

Características Particulares:

Es más flexible que la ejecución cíclica y permite el agregado o eliminación de tareas durante la ejecución.

4.2.3.Prioridad Estática

Fuente:

Real Time Design Patterns: Robust Escalable Architecture For Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:

Sistemas multitarea – Estrategias para la programación de la tarea a ejecutar

Problema:

En S.E. multitarea, algunas tareas son más importantes que otras, en el sentido que tienen requisitos de tiempo de ejecución más estricto, por lo que deben disponer del procesador más veces y/o durante más tiempo que otras tareas.

Solución:

Cuando algunas tareas son más importantes que otras, se genera una jerarquía de tareas, donde las tareas más importantes están en la parte alta de la jerarquía y las

menos importantes en la parte baja. La forma de indicar la ubicación de una tarea dentro de esta jerarquía es a través de su *prioridad*.

De esta manera, el scheduler utiliza la prioridad de cada tarea como criterio para determinar cuál es la siguiente tarea a ejecutar. Este patrón se denomina de prioridad estática porque la prioridad de cada tarea se asigna durante el diseño del sistema y no cambia durante la ejecución del mismo. El diagrama de este patrón se ve en la Figura 18.

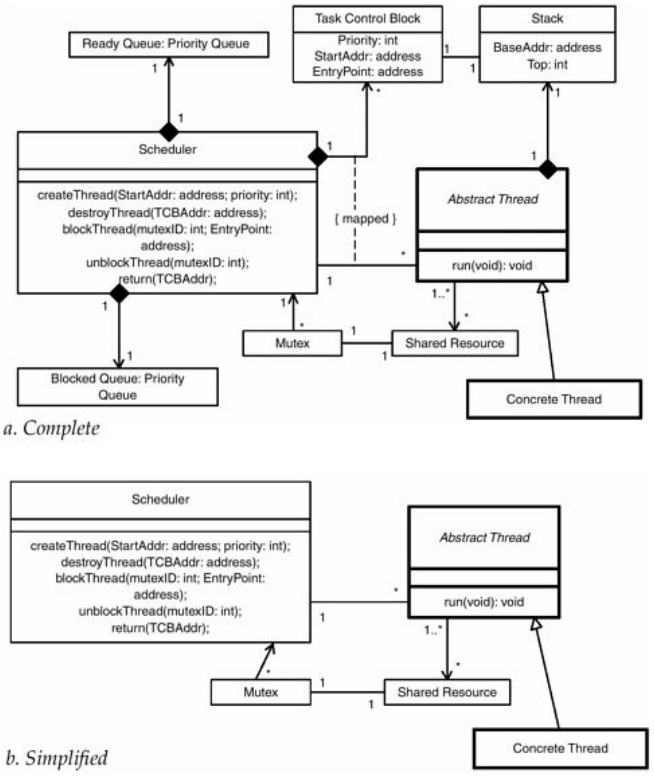

Fig. 18: Prioridad Estática

Los elementos de este patrón son:

- Abstrac Thread: Es una clase abstracta que declara el método *run(),* el cual es usado en cada *Concrete Thread* para invocar su ejecución.
- Concrete Thread: Es el objeto que implementa la funcionalidad de la tarea.
- Scheduler: Es el objeto programador de tareas que inicializa el sistema, carga las tareas e invoca el método *run()* de cada tarea. Las tareas devuelven el control al *Scheduler* al finalizar su actividad o si un recurso que necesitan esta bloquead. El *Scheduler* decide cual es la siguiente tarea a ejecutar según la prioridad de las tareas disponibles para ejecución en la *Ready Queue*.

- Stack: Cada objeto *Concrete Thread* tiene asociado un objeto *Stack* para guardar toda la información asociada a la tarea (su "contexto"), para que al resumir su ejecución, puedan continuar la ejecución en el punto en que se interrumpió.
- Task Control Block (TCB): El *Scheduler* tiene un objeto *Task Control Block* por cada tarea en ejecución, allí guarda información asociada a la tarea, como por ejemplo el punto de ejecución en el que fue interrumpida y la prioridad de la misma.
- Blocked Queue: Es una cola de objetos *TCB*, allí se colocan los *TCB* de las tareas que necesitan acceder a recursos bloqueados. Cuando el recurso se libera, el *TCB* pasa de la *Blocked Queue* a la *Ready Queue*.
- Mutex: es el objeto que arbitra el acceso a los recursos compartidos. Las tareas que necesitan acceder a un recurso compartido verifican su estado, si el *Mutex* está bloqueado, la tarea pasa a estar bloqueada.
- Ready Queue: es una cola de objetos *TCB*, allí se colocan los *TCB* de las tareas disponibles para su ejecución. Cuando en la *Ready Queue* hay una tarea de mayor prioridad que la que se está ejecutando actualmente, esta se detiene, su *TCB* se guarda en la *Ready Queue*, y la tarea de mayor prioridad pasa a ejecutarse.
- Shared resource: Es un elemento que es compartido por varias tareas. Su acceso es controlado por el *Mutex*.

Consecuencias:

Las tareas más importantes (de mayor prioridad) logran mayor uso del procesador. Hay riesgos de bloqueo del sistema si una tarea de menor prioridad bloquea el acceso a un recurso compartido que necesita una tarea de mayor prioridad.

Características Particulares:

Es un esquema simple para lograr que las tareas más importantes tengan mayor tiempo de ejecución. No es flexible si es necesario cambiar las prioridades de las tareas por cambios externos al sistema.

4.2.4.Prioridad Dinámica

Fuente:

Real Time Design Patterns: Robust Escalable Architecture For Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:

Sistemas multitarea – Estrategias para la programación de la tarea a ejecutar

Problema:

En S.E. complejos con varias tareas en ejecución simultáneamente y/o en las que las tareas se crean y destruyen durante la ejecución, no se puede establecer un orden de prioridad para las tareas durante el diseño del sistema.

Solución:

Asignar en forma dinámica la prioridad de las tareas durante la ejecución permite que todas las tareas tengan oportunidad de ser ejecutadas. El diagrama de este patrón se ve en la Figura 19.

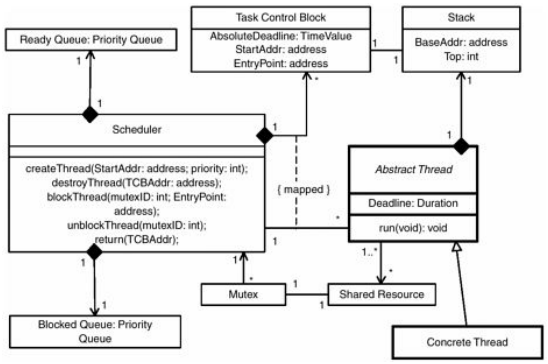

Fig. 19: Prioridad Dinámica

Los elementos de este patrón son:

- Abstrac Thread: Es una clase abstracta que declara el método *run(),* el cual es usado en cada *Concrete Thread* para invocar su ejecución. También contiene el atributo “Deadline”, que indica el tiempo que falta para que se cumpla la meta de temporización de la tarea
- Concrete Thread: Es el objeto que implementa la funcionalidad de la tarea.

- Scheduler: Es el objeto programador de tareas que inicializa el sistema, carga las tareas e invoca el método *run()* de cada tarea. Las tareas devuelven el control al *Scheduler* al finalizar su actividad o si un recurso que necesitan esta bloquead. El *Scheduler* decide cual es la siguiente tarea a ejecutar según la prioridad de las tareas disponibles para ejecución en la *Ready Queue*.
- Stack: Cada objeto *Concrete Thread* tiene asociado un objeto *Stack* para guardar toda la información asociada a la tarea (su "contexto"), para que al resumir su ejecución, puedan continuar la ejecución en el punto en que se interrumpió.
- Task Control Block (TCB): El *Scheduler* tiene un objeto *Task Control Block* por cada tarea en ejecución, allí guarda información asociada a la tarea, como por ejemplo el punto de ejecución en el que fue interrupida y la prioridad de la misma.
- Blocked Queue: Es una cola de objetos *TCB*, allí se colocan los *TCB* de las tareas que necesitan acceder a recursos bloqueados. Cuando el recurso se libera, el *TCB* pasa de la *Blocked Queue* a la *Ready Queue*.
- Mutex: es el objeto que arbitra el acceso a los recursos compartidos. Las tareas que necesitan acceder a un recurso compartido verifican su estado, si el *Mutex* está bloqueado, la tarea pasa a estar bloqueada.
- Ready Queue: es una cola de objetos *TCB*, allí se colocan los *TCB* de las tareas disponibles para su ejecución. Cuando en la *Ready Queue* hay una tarea de mayor prioridad que la que se está ejecutando actualmente, esta se detiene, su *TCB* se guarda en la *Ready Queue*, y la tarea de mayor prioridad pasa a ejecutarse.
- Shared resource: Es un elemento que es compartido por varias tareas. Su acceso es controlado por el *Mutex*.

Consecuencias:

No es posible predecir el orden de ejecución de las tareas. Este patrón es aplicable a tareas que tienen aproximadamente la misma importancia, por lo que el criterio de selección para su ejecución está determinado por el cumplimiento de las metas de temporización de la tarea.

Características Particulares:

Este patrón prioriza la urgencia de una tarea, al modificarle su prioridad en forma dinámica. La prioridad de cada tarea depende del tiempo que falta para que se cumpla su requisito de temporización; las tareas a las que les queda menos tiempo hasta alcanzar su requisito de temporización son las de mayor prioridad. Esto se conoce como “Earliest Deadline First” y se demuestra que es una estrategia óptima para alcanzar los requisitos de temporización de distintas tareas en el sentido que si un grupo de tareas independientes, caracterizadas por un tiempo de arribo y un tiempo máximo de ejecución (deadline) pueden ser programadas para ser ejecutadas por algún algoritmo de tal manera de cumplir los tiempos máximos; entonces estas tareas pueden ser programadas para ser ejecutadas por el algoritmo EDF y serán completadas antes de alcanzar su tiempo máximo de ejecución [19].

Capitulo 5. Patrones para S.E. con requerimientos de seguridad funcional y confiabilidad

5.1. Patrones aplicados a los canales de procesamiento

5.1.1.Canal único protegido

Fuente:

Real Time Design Patterns: Robust Escalable Architecture For Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:

Seguridad funcional sin aplicar redundancia

Problema:

Las mejoras en la seguridad funcional y la confiabilidad de un sistema implican implementar cierta cantidad de redundancia, pero implementar redundancia es costoso, tanto en costos recurrentes (costo por cada unidad al agregar hardware redundante); como en costos de desarrollo (costo de implementar redundancia heterogénea).

Solución:

No todos los sistemas necesitan el nivel de redundancia de los patrones habituales asociados a estos tipos de sistema. Este patrón mejora la seguridad funcional agregando chequeos y acciones adicionales sobre la información de entrada, interna y salida. Hay un solo canal de procesamiento de la información y la seguridad funcional se mejoran mediante el agregado de chequeos en puntos específicos del canal. La estructura de este patrón se ve en la Figura 20 en sus dos variantes: lazo abierto (open loop) y lazo cerrado (closed loop). La diferencia entre ambos es que la variante de lazo cerrado agrega el monitoreo de los resultados de la salida generada por el canal.

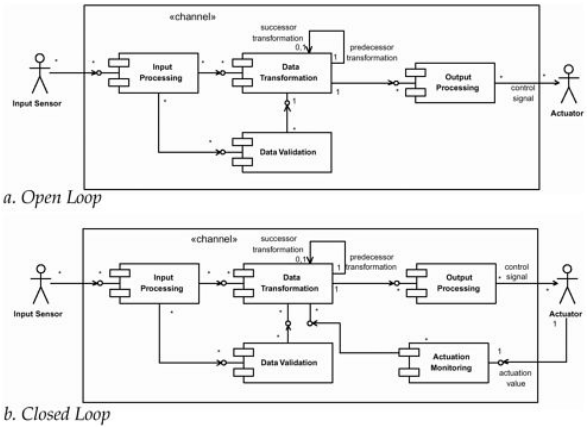

Fig. 20: Canal Único Protegido

Los elementos de este patrón son:

- Actuator: Es el subsistema (hardware o hardware + software) que realiza acciones en función de la salida del canal. Puede haber varios *actuators* controlados por un solo *Channel.*
- Channel: Es el subsistema que adquiere información de entrada del *input sensor* y genera una salida que produce una acción en el *actuator.* Si un dato atraviesa todo el *channel* antes de que se adquiera otro dato, se dice que el canal es seriado (*serial channel*); si el *channel* puede contener varios datos (cada uno de ellos posiblemente en distintos estados de transformación) al mismo tiempo, se dice que es un canal paralelo (*parallel channel*)
- Data transformation: este componente realiza una transformación simple del dato que se está procesando en el canal. Un canal puede tener varios *data transformation*, los cuales están conectados en forma seriada y la salida de uno es la entrada del siguiente. La salida del último *data transformation* del canal pasa a ser la entrada del *output processing*.
- Data Validation: Este componente realiza chequeos sobre distintas partes del *channel* durante el procesamiento de un dato. Estos chequeos pueden ser sobre el hardware del *channel* o sobre los datos internos del mismo, como ser las salidas de los componentes *data transformation*.
- Input processing: este componente toma los datos del *input sensor* y les hace un procesamiento básico (p.ej. escalarlos o quitar información para corrección de errores) antes de pasarlos al componente *data transformation*.

- Input sensor: Este subsistema genera la información que va a procesar el *channel*, puede generar uno o varios datos y su salida puede ser utilizada por uno o varios *channel*.
- Output processing: este componente realiza la última etapa de transformación del dato, convirtiéndolo en algo entendible por el subsistema *actuator*.

Consecuencias:
Este patrón no protege contra fallos persistentes, pero detecta y puede gestionar fallos transitorios, aumentando la confiabilidad. Cuando los componentes Data Validation detectan un fallo, pueden conducir al sistema a un estado seguro, por lo que este patrón aumenta la seguridad funcional del sistema.

Características Particulares:
Es una forma sencilla de aumentar la seguridad funcional y/o la confiabilidad de un canal de procesamiento, ya que monitorea su propio estado interno y la acción de salida.

5.1.2.Monitor – Actuador

Fuente:
Real Time Design Patterns: Robust Escalable Architecture For Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:
Seguridad funcional y confiabilidad de un canal y su actuador.

Problema:
La forma más común de redundancia implica duplicar un canal y su actuador, pero esta solución puede ser demasiado costosa, por otra parte aplicando el patrón *Canal Unico Protegido* no hay redundancia sobre los elementos *Data Validation*, por lo que un fallo en los mismos puede inutilizar el canal y no conducir al sistema al estado seguro en caso de necesidad.

Solución:

Al aplicar redundancia sobre los elementos Data Validation (Data Integrity Checks en este patrón) de un canal, se mejora la seguridad funcional y confiabilidad del mismo. Esto se logra mediante la implementación redundante de un canal paralelo (el Monitoring Channel), que toma los mismos datos que los componentes data validation del canal y puede enviarle señales al canal para indicarle una situación del fallo que el propio canal pudiera no haber detectado. El diagrama de este patrón se ve en la Figura 21.

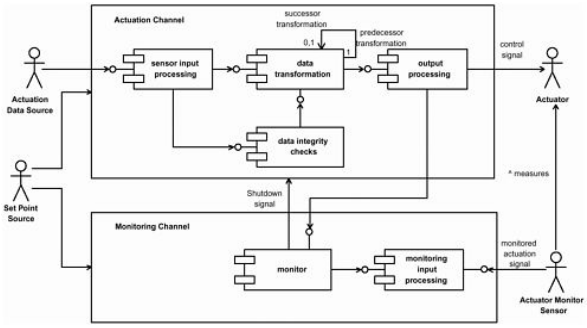

Fig. 21: Monitor - Actuador

Los elementos de este patrón son:

- Actuator: Es el subsistema (hardware o hardware + software) que realiza acciones en función de la salida del *Actuation Channel*. Puede haber varios *Actuators* controlados por un solo canal
- Actuation Channel: Es el subsistema que adquiere información de entrada del *actuator data source* y genera una salida que produce una acción en el *actuator.* Si un dato atraviesa todo el *actuation channel* antes de que se adquiera otro dato, se dice que el canal es seriado (*serial actuation channel*); si el *channel* puede contener varios datos (cada uno de ellos posiblemente en distintos estados de transformación) al mismo tiempo, se dice que es un canal paralelo (*parallel actuation channel*)
- Data transformation: este componente realiza una transformación simple del dato que se está procesando en el canal. Un canal puede tener varios *data transformation*, los cuales están conectados en forma seriada y la salida de uno es la entrada del siguiente. La salida del último *data transformation* del canal pasa a ser la entrada del *output processing*.
- Data Integrity Checks (*Data Validation*): Este componente realiza chequeos sobre distintas partes del *actuation channel* durante el procesamiento de un dato.

Estos chequeos pueden ser sobre el hardware del *actuation channel* o sobre los datos internos del mismo, como ser las salidas de los componentes *data transformation*.

- Sensor Input processing: este componente toma los datos del *actuation data source* y les hace un procesamiento básico (p.ej. escalarlos o quitar información para corrección de errores) antes de pasarlos al componente *data transformation*.
- Actuation Data Source: Este subsistema genera la información que va a procesar el *actuation channel*, puede generar uno o varios datos y su salida puede ser utilizada por uno o varios *actuation channel*. Sus elementos son independientes del *actuator monitor sensor.*
- Output processing: este componente realiza la última etapa de transformación del dato, convirtiéndolo en algo entendible por el subsistema *actuator*.
- Actuator monitor sensor: es el subsistema que monitorea al *actuator* y genera la señal de entrada del *monitoring channel.* Sus elementos son independientes del *actuation data source*
- Monitor: este componente compara la salida esperada (la cual es provista por el *set point source*) con la salida generada (tomada del *actuator monitor sensor* tras ser procesada por el componente *monitoring input processing*) y en función de la comparación, le envía una señal al *actuation channel* para que el mismo pase a su estado seguro.
- Monitoring Channel: este canal implementa la redundancia de chequeo. Recibe la señal de salida esperada del *set point source* y la compara con los valores obtenidos del *actuator*.
- Monitoring Input Processing: este componente toma los datos del *actuator* y les hace un procesamiento básico (p.ej. escalarlos o quitar información para corrección de errores) antes de pasarlos al componente *monitor*.
- Set Point Source: este subsistema genera las señales de comando del *actuation channel* y las señales esperables del *actuator*, a fin de que el *monitoring channel* pueda detectar una diferencia entre la acción esperada del *actuator* y la acción real producida.

Consecuencias:

Esta redundancia permite que si hay un fallo no detectado en el canal, el *monitoring channel* la detecta y se la informa al canal para que conduzca al sistema al estado seguro, aumentando la seguridad funcional; y si hay un fallo en el *monitoring channel*, el canal principal continua funcionando, aumentando la confiabilidad del sistema, ya que continua funcional aun habiendo sucedido un fallo.

Características Particulares:

El *monitoring channel* debe tener en cuenta los retardos y variaciones entre la acción que mide proveniente del *actuator* y el valor de referencia del *set point source* cuando hay un cambio del mismo; a fin de no detectar un falso positivo, que sería estimar que hay una diferencia entre el valor esperado de la acción del *actuator* y la referencia del *set point source* cuando en realidad esta aparente diferencia se debe a la medición de la acción del *actuator* y/o al retardo entre la variación en el *set point source* y su efecto en la salida del *actuator*.

Hay una variación de este patrón, que es cuando el *monitoring channel* no accede al *set point source*, sino que solamente estima si la acción del *actuator* es “razonable”, es decir, si esta dentro de rangos preestablecidos, si sigue una secuencia determinada, etc; pero sin información de qué es lo que debería estar generando el *actuation channel*. A este patrón se lo denomina *“Sanity Check”.*

5.1.3.Watchdog

Fuente:

Real Time Design Patterns: Robust Escalable Architecture For Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:

Seguridad funcional y confiabilidad sin aplicar redundancia

Problema:

En los S.E. con requerimientos estrictos de tiempo real, se considera una falla el no realizar un proceso computacional dentro de un tiempo preestablecido, por lo que es necesario un mecanismo para detectar esta situación.

Solución:

Un componente de software que tiene información de temporización es notificado cuando termina el proceso computacional. Este componente verifica que esta notificación haya llegado antes de que finalice el tiempo establecido; si esto no sucede al cumplirse el tiempo máximo establecido para el proceso computacional, el componente notifica al canal que un fallo ha ocurrido para que el canal lleve al sistema a un estado seguro o realice las acciones necesarias frente al fallo. Un esquema de este patrón se ve en la Figura 22.

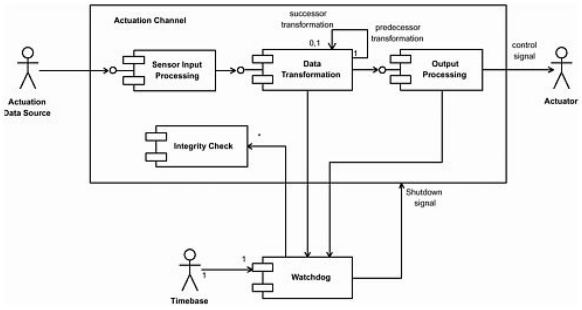

Fig. 22: Watchdog

Los elementos de este patrón son:

- Actuation Channel: Es el subsistema que adquiere información de entrada del *actuator data source* y genera una salida que produce una acción en el *actuator.* Si un dato atraviesa todo el *actuation channel* antes de que se adquiera otro dato, se dice que el canal es seriado (*serial actuation channel*); si el *channel* puede contener varios datos (cada uno de ellos posiblemente en distintos estados de transformación) al mismo tiempo, se dice que es un canal paralelo (*parallel actuation channel*)
- Actuation Data Source: Este subsistema genera la información que va a procesar el *actuation channel*, puede generar uno o varios datos y su salida puede ser utilizada por uno o varios *actuation channel*. Sus elementos son independientes del *actuator monitor sensor.*
- Actuator: Es el subsistema (hardware o hardware + software) que realiza acciones en función de la salida del canal. Puede haber varios *actuators* controlados por un solo canal
- Data transformation: este componente realiza una transformación simple del dato que se está procesando en el canal. Un canal puede tener varios *data transformation*, los cuales están conectados en forma seriada y la salida de uno

es la entrada del siguiente. La salida del último *data transformation* del canal pasa a ser la entrada del *output processing*.

- Data Integrity Checks (*Data Validation*): Este componente realiza chequeos sobre distintas partes del *actuation channel* durante el procesamiento de un dato. Estos chequeos pueden ser sobre el hardware del *actuation channel* o sobre los datos internos del mismo, como ser las salidas de los componentes *data transformation*.
- Output processing: este componente realiza la última etapa de transformación del dato, convirtiéndolo en algo entendible por el subsistema *actuator*.
- Sensor Input processing: este componente toma los datos del *actuation data source* y les hace un procesamiento básico (p.ej. escalarlos o quitar información para corrección de errores) antes de pasarlos al componente *data transformation*.
- Timebase: Es una fuente independiente de temporización que utiliza el *watchdog* como referencia de tiempo.
- Watchdog: este elemento espera la señal de finalización del proceso computacional proveniente del *actuation channel*, si la misma no ocurre antes del tiempo preestablecido (el se verifica con la información del *timebase*), el *watchdog* indica el fallo al componente *integrity check* para que realice las acciones apropiadas.

Consecuencias:

Este patrón es específico para detectar fallos en la temporización del procesamiento de datos del canal. Tambien sirve para detectar deadlocks y/o bloqueos (hangs) en el canal, ya que cuando se producen, se deja de enviar la señal de finalización al *watchdog*.

Características Particulares:

En la implementación de este patrón se requiere una base de tiempo independiente del canal; dicha base de tiempo normalmente se implementa en un hardware separado.

Una variante de este patrón requiere que la señal de finalización contenga información acerca de a que parte del proceso computacional corresponde, a fin de detectar que estas señales no solo aparezcan en el tiempo requerido, sino que también sucedan en

la secuencia apropiada. A este patron se lo denomina *"keyed watchdog"* o *"sequential watchdog",* y permite detectar casos de deadlock en los cuales el canal sigue enviando la señal de finalización, la cual al no cumplir la secuencia (debido a la situación de deadlock), permite la detección del fallo.

5.1.4.Gestor de Seguridad

Fuente:

Real Time Design Patterns: Robust Escalable Architecture For Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

Área de Aplicación:

Seguridad funcional y confiabilidad con redundancia

Problema:

Algunas veces, frente a un fallo del sistema, las acciones a realizar son muy complejas, diferentes según el tipo de fallo, y/o deben realizarse en una secuencia específica, por lo que es necesario coordinar y verificar la realización de estas acciones.

Solución:

Se implementa un componente del sistema cuya función es, frente a un fallo, coordinar y verificar las acciones necesarias, las cuales pueden ser diferentes según el origen y tipo de fallo. A este componente se lo denomina "Gestor de Seguridad" (Safety Executive). Un Gestor de seguridad puede tener a su cargo a varios canales de procesamiento, un watchdog para verificar requerimientos de temporización en las acciones a tomar y opcionalmente un canal de ejecución independiente (fail safe processing channel) que implementa las acciones frente a un fallo. El diagrama de este patrón se ve en la Figura 23.

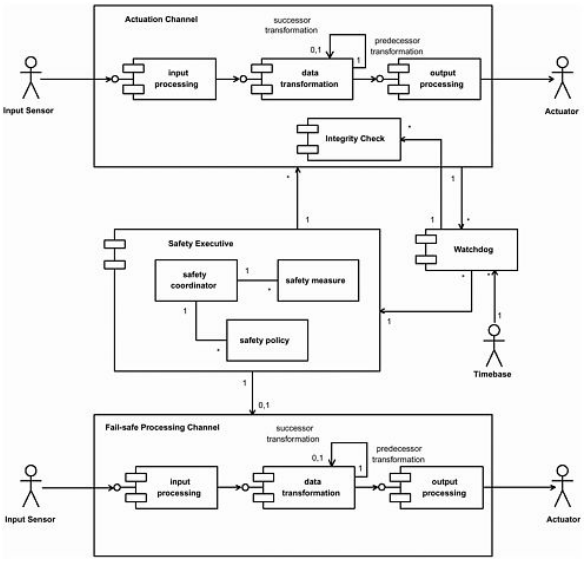

Fig. 23: Gestor de Seguridad

Los elementos de este patrón son:

- Actuation Channel: Es el subsistema que adquiere información de entrada del *actuator data source* y genera una salida que produce una acción en el *actuator.* Si un dato atraviesa todo el *actuation channel* antes de que se adquiera otro dato, se dice que el canal es seriado (*serial actuation channel*); si el *channel* puede contener varios datos (cada uno de ellos posiblemente en distintos estados de transformación) al mismo tiempo, se dice que es un canal paralelo (*parallel actuation channel*)
- Input Sensor (Actuation Data Source): Este subsistema genera la información que va a procesar el *actuation channel*, puede generar uno o varios datos y su salida puede ser utilizada por uno o varios *actuation channel*.
- Actuator: Es el subsistema (hardware o hardware + software) que realiza acciones en función de la salida del canal. Puede haber varios *actuators* controlados por un solo canal
- Data transformation: este componente realiza una transformación simple del dato que se está procesando en el canal. Un canal puede tener varios *data transformation*, los cuales están conectados en forma seriada y la salida de uno es la entrada del siguiente. La salida del último *data transformation* del canal pasa a ser la entrada del *output processing*.
- Fail safe Processing Channel: Es el canal que realiza las acciones sobre el sistema cuando el *safety executive* determina que hubo un fallo. Es similar al *actuation*

channel, excepto que no tiene *data integrity checks*, ya que actúa cuando ocurrió un fallo.

- Data Integrity Checks (*Data Validation*): Este componente realiza chequeos sobre distintas partes del *actuation channel* durante el procesamiento de un dato. Estos chequeos pueden ser sobre el hardware del *actuation channel* o sobre los datos internos del mismo, como ser las salidas de los componentes *data transformation*.
- Input processing: este componente toma los datos del *actuation data source* y les hace un procesamiento básico (p.ej. escalarlos o quitar información para corrección de errores) antes de pasarlos al componente *data transformation*.
- Output processing: este componente realiza la última etapa de transformación del dato, convirtiéndolo en algo entendible por el subsistema *actuator*.
- Safety coordinator: este componente es el encargado de coordinar las acciones, de acuerdo al tipo de fallo, las cuales se implementan en las *safety measure*.
- Safety executive: es el subsistema encargado de gestionar las acciones ante la ocurrencia de un fallo.
- Safety measure: Es un componente que establece las acciones a realizar de acuerdo al fallo ocurrido. Hay un componente *safety measure* para cada tipo de fallo que puede gestionar el *safety executive*.
- Safety policy: es el componente que establece los criterios con que deben aplicarse las safety measure ante la ocurrencia de fallo, ya que el mismo puede tener que gestionarse de distintas maneras de acuerdo al estado del sistema al momento del fallo.
- Timebase: Es una fuente independiente de temporización que utiliza el *watchdog* como referencia de tiempo.
- Watchdog: este elemento espera la señal de finalización del proceso computacional proveniente del *actuation channel*, si la misma no ocurre antes del tiempo preestablecido (el cual se verifica con la información del *timebase*), el *watchdog* indica el fallo al componente *integrity check* para que realice las acciones apropiadas.

Consecuencias:
Este patrón gestiona de manera completa las acciones a realizar en caso de producirse un fallo en sistemas complejos. Como contrapartida, su implementación es también compleja ya que se deben implementar varios componentes y subsistemas.

Características Particulares:
Al tener componentes *safety measure* y *safety policy*, este patrón es altamente configurable para contemplar nuevos fallos o distintas estrategias de acción frente a la ocurrencia de los mismos.

5.2. Patrones aplicados a los datos

5.2.1. Complemento a uno

Fuente:
Design Patterns for Embedded Systems in C: An Embedded Software Engineer Toolkit, Bruce Powel Douglas, Elsevier, 2010

Área de Aplicación:
Seguridad funcional y confiabilidad en los datos con redundancia

Problema:
Los datos almacenados en memoria (tanto volátil como no volátil) son suceptibles de sufrir corrupción por fallos en el hardware, en la alimentación del sistema, o por causas externas (p.ej. ruido electromagnético)

Solución:
Si al almacenar un dato binario se almacena también su complemento (todos los bits invertidos), al acceder al mismo se lo puede comparar con su complemento y detectar fallos en uno o más bits. El diagrama de este patrón se puede ver en la Figura 24.

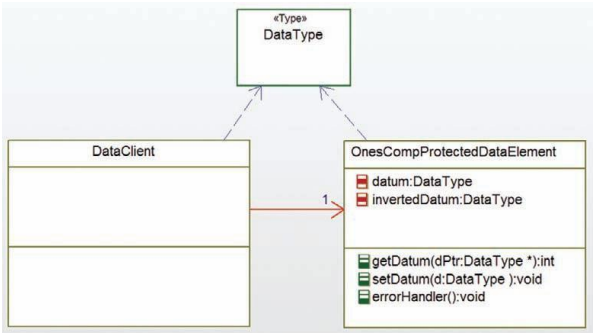

Fig. 24: Complemento a Uno

Los elementos de este patrón son:

- DataClient: Es el elemento que lee o escribe el dato protegido
- OnesCompProtectedDataElement: Es el componente que combina el almacenamiento del dato y su complemento, junto con servicios para detectar si ha habido corrupción de datos. Cuando se invoca a *setDatum()*, el dato se guarda como se recibe y en forma invertida; cuando se invoca a *getDatum()* se comparan el dato con su versión invertida, y en caso de encontrarse un fallo se invoca a *errorHandler()*, quien tomara las acciones correspondientes.

Consecuencias:

El espacio físico de almacenamiento se reduce a la mitad, ya que los datos se almacenan por duplicado; y hay una penalización de performance, ya que cada acceso a datos implica una comparación (lectura) o la generación del dato invertido (escritura)

Características Particulares:

Este patrón es apropiado para pequeños grupos de datos críticos, ya que si la cantidad de datos es grande, la penalización en el acceso puede ser alta y hacerlo inaplicable en S.E. con requerimientos de tiempo real. Se puede mejorar la capacidad de detección de fallos si el dato y su complemento se almacenan en dispositivos de memoria físicamente independientes

5.2.2.CRC

Fuente:

Design Patterns for Embedded Systems in C: An Embedded Software Engineer Toolkit, Bruce Powel Douglas, Elsevier, 2010

Área de Aplicación:

Seguridad funcional y confiabilidad en los datos

Problema:

Los datos almacenados en memoria (tanto volátil como no volátil) son suceptibles de sufrir corrupción por fallos en el hardware, en la alimentación del sistema, o por causas externas (p.ej. ruido electromagnético). En algunos casos el volumen de datos es tan grande respecto al espacio físico disponible que no es viable aplicar una redundancia total sobre los mismos.

Solución:

Un esquema de chequeo de redundancia cíclica (CRC – Ciclic Redundancy Check) sobre un conjunto de datos consiste en generar un “valor de chequeo” (CRC Value) en base a esos datos aplicando un determinado algoritmo. Dicho valor de chequeo se adiciona al conjunto de datos. Para verificar la integridad de los mismos, se vuelve a aplicar el algoritmo sobre el conjunto de datos y se verifica si el valor de chequeo obtenido coincide con el valor de chequeo asociado a los datos. Si este no coincide, se está en presencia de errores en los datos. Se considera un error individual en los datos el que haya un bit invertido en alguno de sus elementos; es decir, si hay dos bits invertidos en algún lugar del conjunto de datos, se considera que hubo dos errores. El esquema de este patrón se puede ver en la Figura 25.

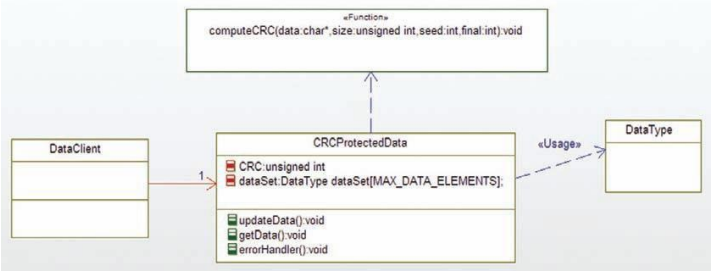

Fig. 25: CRC

Los elementos de este patron son:

- computeCRCData: este componente realiza el cálculo del valor de chequeo del conjunto de datos
- CRCProtectedData: es el conjunto de datos junto con su valor de chequeo, cuando se invoca a *updateData()* se agrega un nuevo dato al conjunto y se vuelve a calcular el valor de chequeo. Cuando se invoca a *getData()*, se verifica que el valor de chequeo del conjunto de datos coincida con el valor de chequeo asociado; si esto no sucede, se invoca a *getErrorHandler()*, quien realiza las acciones adecuadas.
- DataClient: es el elemento que lee o escribe en el conjunto de datos.
- DataType: es el tipo de dato básico sobre el que se va a aplicar este patrón.

Consecuencias:

Al aplicar este patrón es posible proteger grandes volúmenes de datos con un costo bajo en espacio de memoria y acceso a los mismos. Por otra parte mediante los distintos tipos de algoritmos aplicables es posible realizar diferentes compromisos entre el espacio adicional en memoria para los valores de chequeo, la sobrecarga computacional de la verificación de los datos, y la capacidad del algoritmo de recuperar la información frente a una cierta cantidad de errores en los datos.

Características Particulares:

Los algoritmos para generar el valor de chequeo son de variado tipo; algunos permiten detectar hasta una cierta cantidad de errores, otros permiten detectar y corregir hasta cierta cantidad de errores, y en otros la capacidad de detección y corrección de errores depende de si los mismos suceden aleatoriamente o en una secuencia contigua dentro del conjunto de datos. En el caso particular en que el algoritmo solo puede detectar si hubo uno o más errores en el conjunto de datos, se dice que el valor de chequeo es una “firma” o “hash” del conjunto de datos.

5.2.3.Datos Inteligentes

Fuente:

Design Patterns for Embedded Systems in C: An Embedded Software Engineer Toolkit, Bruce Powel Douglas, Elsevier, 2010

Área de Aplicación:

Seguridad funcional y confiabilidad en los datos

Problema:

Muchas veces es necesario que los datos cumplan con ciertos requerimientos: que su valor esté dentro de un rango preestablecido, que su tamaño no exceda el espacio de memoria asignado para ellos, en el caso de datos indexados que el índice no exceda la cantidad de datos disponibles, etc.

Solución:

Al implementar las funciones de acceso a los datos, se realizan chequeos para verificar que el contenido de los mismos y su forma de acceso cumple con los requerimientos preestablecidos y se establece un mecanismo para informar el requerimiento no cumplido. El diagrama de este patrón se ve en la Figura 26.

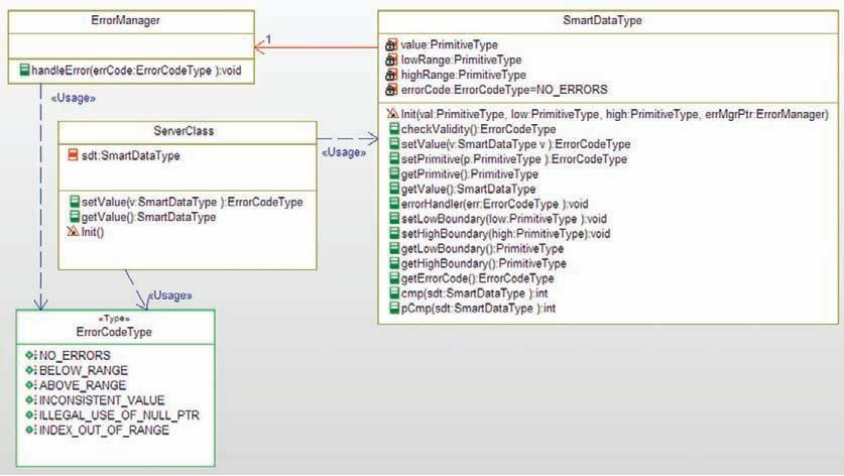

Fig. 26: Datos Inteligentes

Los elementos de este patrón son:

- ErrorCodeType: este componente codifica el tipo de prerrequisito que no se está cumpliendo.
- ErrorManager: este componente gestiona las acciones a realizar cuando el dato o su acceso no cumple con algún prerrequisito.
- ServerClass: es el elemento que lee o escribe el dato.
- SmartDataType: es el componente que integra el dato junto con métodos que implementan la verificación del cumplimiento de los prerrequisitos.

Consecuencias:

Aplicar este patrón genera una sobrecarga durante el acceso, ya que el dato es chequedo para verificar el cumplimiento de sus prerrequisitos. La ventaja es que este chequeo es automático y responsabilidad del dato en sí mismo, no de los productores o consumidores del mismo.

Características Particulares:

Este patrón es una generalización del “Smart Pointer”, ya que lo generaliza a cualquier tipo de dato, no solo a punteros.

Capitulo 6. Conclusiones

La ingeniería de software aplicada al desarrollo de S.E. ha alcanzado un nivel de madurez tal que permite disponer de técnicas y herramientas aplicables a esta área de desarrollo. En particular el uso de patrones, debido a que permite reutilizar soluciones ya probadas, lo que contribuye a mejorar el desarrollo de software para S.E. en general y S.E. con requerimientos de tiempo real, seguridad funcional y confiabilidad. A su vez estos patrones son aplicables a distintos aspectos del desarrollo: arquitectura, concurrencia, gestión de la memoria, verificación formal de requerimientos, gestión de las tareas, redundancia en el cómputo, integridad de los datos, etc.

Aun así, es necesario profundizar la gestión de los requerimientos no funcionales a lo largo de todo el ciclo de desarrollo del software para S.E., tanto en los requerimientos como en la arquitectura, implementación y testing de estos sistemas; ya que los requerimientos no funcionales muchas veces condicionan a los requerimientos funcionales, como por ejemplo el requerimiento de memoria, consumo y/o costo del hardware vs qué algoritmo de compresión utilizar al implementar un sistema de audio y video.

Por otra parte el desarrollo de las plataformas de computo aplicadas a sistemas embebidos (el hardware) está evolucionando de sistemas multiprocesadores simétricos (sistemas multiprocesadores en los que todos los procesadores son iguales) a sistemas multiprocesadores asimétricos, en los cuales no todos los procesadores tienen las mismas prestaciones, sino que algunos procesadores están optimizados para alta capacidad de computo, y otros procesadores están orientados a requerimientos de tiempo real o bajo consumo. Estas nuevas arquitecturas de hardware condicionan la implementación de soluciones de software por lo que es de esperar que generen patrones específicos de aplicación al desarrollo de software sobre estas plataformas, por lo que se propone para un futuro trabajo la identificación de patrones aplicables a S.E. cuya plataforma de cómputo este implementada con multiprocesadores asimétricos.

Capitulo 7. Referencias

[1] http://www.barrgroup.com/Embedded-Systems/Glossary-E (2007-11-19, verificado 2015-04-28)

[2] “Programming Embedded Systems with C and GNU Development Tools” 2nd Edition, Michael Barr, Anthony Massa, O’Reilly Media, 2006

[3] http://en.wikipedia.org/wiki/Embedded_system (verificado 2015-04-28)

[4] Embedded Software Engineering: The State of the Practice. Bas Graaf, Marco Lormans, Hans Toetenel. IEEE Software Volume 20 Issue 6 Nov. 2003

[5] Embedded Software: Facts, Figures and Future. Christof Ebert, Capers Jones, IEEE Computer Volume 42 Issue 4 Abril 2009

[6] A Pattern Languaje: Towns, Buildings , Construction: Christofer Alexander, 1977

[7] Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison Wesley, 1994

[8] “Software Engineering” 9th Ed. Ian Sommerville, Addison Wesley, 2011 (p.538)

[9] Real Time Specification Patterns, Sascha Konrad, Betty Cheng, ICSE05, Mayo de 2005

[10]Patterns in Property Specifications for Finite State Verification, Matthew Dwyyer, George Avrunin, James Corbett, ICSE’99, mayo de 1999

[11] Specifying real time properties with metric temporal logic, R. Koymans, Journal of Real Time Systems Volume 2, Issue 4 pags.255-299, Nov.1990

[12] Techniques for automatic verification of real time systems, R.Alur, PhD Thesis, Stanford University, 1991

[13] A graphical enviroment for the design of concurrent real time systems, Moser, Ramakrishna, Kutty, Meliar-Smith, Dillon, ACM Transactions on Software Engineering and Methodology Volume 6 Issue 1 pag:31-79, 1997

[14] Real Time Design Patterns: Robust Escalable Architecture For Real Time Systems. Bruce Powell Douglas, Addison Wesley 2002

[15] “Software Engineering” 9th Ed. Ian Sommerville, Addison Wesley, 2011 (539)

[16] Design Patterns for Embedded Systems in C: An Embedded Software Engineer Toolkit, Bruce Powel Douglas, Elsevier, 2010

[17] Design Pattern Representation for safety critical embedded systems, Ashraf Armoush, Falk Salewsky, Stefan Kowalewski, J.Software Engineering & Applications 2009 2:1-12, Abril de 2009

[18] IEC61508 Functional Safety for electrical/electronic/programmable electronic safety related systems, International Electrotechnical Comission, 1998

[19] Schedulling Algorithm for Multiprogramming in a Hard Real Time Environment, C.L: Liu, James Layland, Journal of ACM Volume 20 Issue 1, pag:46-61, 1973