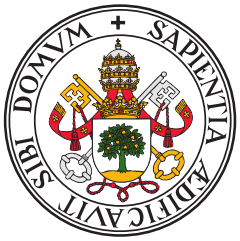

Universidad de **Valladolid**

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Mención en Tecnologías de la Información

DESARROLLO DE UN *Kernel RTOS* SEGURO PARA PROCESADORES *ARM® Cortex-R*

AUTOR:

MANUEL ADRIÁN CASTILLO MEJÍAS

TUTOR:

JESÚS ARIAS ÁLVAREZ

AGRADECIMIENTOS

Mi gratitud a Jesús Arias por despertar este interés en el mundo de los sistemas empotrados y microcontroladores, completamente desconocido para mí, y aceptar mi propuesta de Trabajo de Fin de Grado. También por guíarme, atenderme y asesorarme en el apartado teórico. Asimismo, a Yania Crespo por atender mis peticiones, ya que su ayuda en el campo de la Ingeniería del *Software* me permitió continuar con la elaboración del proyecto en un momento crítico.

Agradecer ante todo a Alberto, capaz de sacar lo mejor de mí mismo e inspirarme, ya que sin él en los peores momentos yo no habría sido capaz de reponerme ni tampoco de completar este trabajo. También a Jose y a Dani por estar ahí cuando lo he necesitado. A Cristina por todo su apoyo pese a todo.

A mi familia por su soporte, dado que sin ellos yo no estaría hoy aquí.

RESUMEN

Los sistemas empotrados basados en *microcontrolador* se encuentran presentes en un amplio rango de dominios de aplicación, tales como dispositivos electrónicos de consumo, dispositivos médicos, automoción o aeronáutica entre otros, donde suscitan especial interés aquellos que imponen requisitos y restricciones tales como seguridad, alta disponibilidad, redundancia, concurrencia, baja latencia y recuperación frente a fallos, dada su especificidad. Este tipo de sistemas de información presentan como principal característica una fuerte limitación en la disposición de recursos *hardware*, lo que obliga al minucioso estudio de las necesidades que las tareas requieren de ellos. Así pues, se benefician de procesos y metodologías de elaboración *software* que tienen como objetivo abordar y manejar las imposiciones de su entorno y sus características, así como la complejidad asociada al problema para el que ofrecen solución.

La disposición de procesos de manera concurrente supone un especial desafío para este tipo de sistemas, en tanto en cuanto estos comparten entre otros aspectos el espacio de direcciones físicas de la plataforma. Bajo dicho espacio se dispone la memoria principal, cuyo correcto funcionamiento se considera indispensable, puesto que albergar objetos dinámicos, así como parte de la lógica que constituye a la aplicación. Por ello se demanda la existencia de una entidad, que recibe el nombre de *kernel*, capaz de ejercer control sobre el uso que reciben tales recursos, a la par que supervisa las operaciones que afectan a aquellas secciones consideradas críticas. Además de este mecanismo, es posible insertar técnicas de protección que actúan en tiempo de ejecución, hecho que supone un detrimento en la latencia de las operaciones. Por contra, otras incorporan técnicas de carácter estático, las cuales no tienen impacto sobre el rendimiento, pero requieren de herramientas de construcción *software* y lenguajes de programación específicos para ello. Es posible ofrecer una solución agnóstica de la plataforma y con una mínima huella en la ejecución como combinación entre el *kernel* y un mecanismo *hardware* específico al que se conoce como unidad de protección de memoria o *MPU* (*Memory Protection Unit*). Este, dada su simplicidad, consigue aislar los recursos necesarios por proceso sin necesidad de establecer esquemas de traducción de direcciones y espacios lógicos –propios de sistemas de cómputo general–, manteniendo el soporte para características de protección estáticas o dinámicas si así se requiriese.

En este trabajo se realiza un estudio de la funcionalidad ofrecida por una gama de implementaciones de unidad de protección disponibles en el mercado, agrupando aquellas características comunes bajo las denominadas operaciones primitivas. Asimismo, se analizará, diseñará e implementará un prototipo que incorpore tales operaciones, con objeto de evaluar la eficacia de una de estas técnicas de protección para la arquitectura ARMv7-R, que será expuesta a accesos no previstos, de gran impacto en la plataforma y su entorno. Para este fin se propone un *kernel* básico que provee una serie de operaciones para la configuración, asignación y gestión estática de recursos por proceso. Se aporta además una metodología de desarrollo *software* que logra cierta autonomía respecto de las herramientas y bibliotecas propietarias, sustituidas por alternativas *software* de código abierto, así como consigue disminuir el acomplamiento que presenta la aplicación respecto de la infraestructura, primando la portabilidad.

ÍNDICE GENERAL

CAPÍTULO 1

INTRODUCCIÓN

1.1 CONTEXTO

Frente a los sistemas de cómputo general –como ordenadores de uso personal o los teléfonos inteligentes–, se encuentran los sistemas empotrados o embebidos cuya característica principal es la integración física en procesos industriales, mecánicos u otros sistemas informáticos. Los sistemas empotrados son considerados un agregado informático completo, bajo el cual pueden identificarse todos los elementos lógicos de un computador miniaturizado, como son: unidad central de procesamiento, memoria principal, periféricos y almacenamiento persistente; tal que constituyen la infraestructura *hardware*. Su principal función consiste en asegurar una ejecución sin contratiempos para su marco de aplicación, tal que ofrece un conjunto de operaciones que constituyen el *software* embebido y se adecuan a los requisitos y restricciones que le son impuestos. [6]

En la actualidad, los avances tecnológicos en procesos de fabricación de circuitos integrados, permiten cubrir las crecientes demandas en capacidad de cómputo que exigen los sistemas de información, por lo que es posible incorporar nuevos módulos lógicos dedicados generando así un amplio catálogo de sistemas empotrados que cubren las necesidades específicas de los dominios de aplicación. Gracias a esta mejora de rendimiento se posibilita la creación de nuevos servicios y operaciones, aunque por contra, esto puede introducir puntos de fallo en el sistema no presentes con anterioridad, que sumado a las imposiciones que establece el entorno desemboca en un incremento en la complejidad de desarrollo de estos sistemas. Se deberán escoger las metodologías de desarrollo adecuadas que aborden los problemas presentes, de forma que la aplicación resultante de este proceso haga debido uso de los recursos disponibles y cumpla con las limitaciones que padece.

Los sistemas operativos se encargan de administrar permisos en operación, de gestionar la recursos disponibles, así como de permitir desacoplar la plataforma subyacente de las tareas de usuario que conforman la aplicación, entre otras funciones. Se sirven con este fin de mecanismos especializados, que por ejemplo permiten la traducción de direcciones lógicas al espacio de direcciones físicas, de manera que cada proceso recibe su propio espacio estanco y comprueba los permisos en operación para cada uno de ellos. Sin embargo, dada la coerción que ya de por sí presentan los sistemas empotrados, son inviables este tipo de mecanismos, en la medida en que por un lado conlleva un incremento en el coste de producción difícilmente asumible, debido a la lógica necesaria para su configuración, en cuanto que se incorporan a otro sistema de información –donde generalmente se centra el gasto–, mientras que por otro, provocan un aumento en latencias de tiempo de respuesta y por ende una reducción del rendimiento de la plataforma. Es por ello que estos sistemas ofrecen aproximaciones dedicadas para la protección, que serán objeto de estudio a lo largo de este trabajo.

1.2 MOTIVACIÓN

En la actualidad, el uso de elementos *hardware* para la protección de memoria, cuya gestión se delega al *kernel*, resulta la tónica habitual para los considerados sistemas de cómputo general y sistemas empotrados de alto rendimiento. En particular destaca la *MMU*, en tanto en cuanto ha desplazado a otro tipo de técnicas, dada su relativa simplicidad a la hora de su configuración y despliegue. Esta realiza una doble función, donde por un lado permite la extensión de la memoria disponible en la plataforma –mediante la traducción de direcciones–, proporcionando así una capacidad máxima aparente, la cual difiere de la real, mientras que por otro, también habilita la comprobación de permisos en acceso, de manera que sólo la tarea propietaria de un recurso es capaz de operar sobre este. Además, este mecanismo destaca por su capacidad para desacoplar el espacio en el que reside una tarea de la tecnología subyacente –por ejemplo, los objetos que la conforman pueden residir indistintamente memoria volátil, no volátil o en ambas–. Así, acciones tales como el intercambio de fragmentos de la aplicación bajo demanda –si un objeto resulta alterado/corrupto parcialmente o totalmente–, la restauración del estado de una tarea en caso de fallo o la definición de direcciones aleatorias para el alojamiento de secciones tras la planificación –en cuyo caso no sería posible la recuperación frente a fallos pero consigue aumentar la protección frente a atacantes–, resultan transparentes y consiguen un equilibrio entre la estabilidad y la seguridad de la ejecución.

En lo que compete a los sistemas empotrados basados en microcontrolador, la ausencia de estas capacidades constriñe en gran medida la ejecución de la aplicación, así como dificulta su recuperación ante condiciones adversas –tanto exógenas como endógenas–, cuyos efectos pueden incluso propagarse al entorno en el que se insertan. Sin embargo, dado que resulta habitual encontrar, para este tipo de propuestas, una definición conocida de antemano en lo que respecta al flujo concreto de la solución –esto es, se fija estáticamente–, es posible identificar algunas de las amenazas, a la par que se habilita una evaluación de su impacto, así como el alcance de sus consecuencias. Una vez patentes podrían anticiparse y así evitar su aparición o en su defecto mitigar su acción –un fallo queda a lo sumo contenido en el espacio asociado a la tarea que lo propició–, mediante la inserción de técnicas *software* para la protección con carácter estático y dinámico. No obstante, estos mecanismos seguirían presentando vulnerables al constituirse como parte integral de la propia lógica, por lo que dadas las implicaciones de estos hechos, la búsqueda de una alternativa en este tipo de contextos se erige como una prioridad. Tal es así, que algunos proveedores incorporan a sus productos mecanismos adicionales para facilitar el desempeño de labores de protección –de manera homóloga a otros sistemas de computación–, soluciones de entre las que destacan aquellas que tienen en su haber a la unidad de protección de memoria. A pesar de ello y en contra de lo que cabría esperar, el uso de técnicas de protección basadas en *MPU* resulta meramente anecdótico. Por norma general se subestiman e infravaloran sus capacidades con respecto a los requerimientos de la aplicación, mientras que en otros casos, estos se perciben erróneamente como una sobrecarga a una ya lastrada –por ejemplo, dada una pobre optimización de la lógica– ejecución.

Por ello, se pretende a lo largo de este trabajo demostrar el beneficio que supone el establecimiento de regiones estáticas, tal que cubren para su protección secciones de la aplicación consideradas como críticas, logrando la estabilidad de la plataforma sin devaluación alguna en su rendimiento y capacidades. En definitiva, se pretende bajo este documento exponer las bondades de este mecanismo con intención de mejorar su adopción.

1.2.1 Alcance del proyecto

Se pretende establecer el carácter para la aplicación ofrecida como solución al problema a tratar, en cuanto que este determinará las decisiones tomadas durante las distintas fases del desarrollo, que afectan al trabajo necesario y su planificación asociada, así como el resultado final que se pretende conseguir. Para este proyecto se concibe el producto a elaborar como un mero prototipo, el cual permitirá la evaluación y el examen de las bondades de los mecanismos de protección, bajo un punto de vista de la estabilidad de la plataforma. Tal interpretación se basará en supuestos que únicamente evidencien las características de tales técnicas de protección y no tomará como referencia la construcción de un producto final y completamente funcional, bajo los que se atienden necesidades concretas del cliente y el entorno. No obstante, cabe la posibilidad de plantear algunos elementos propios de estos últimos con carácter futuro, facilitando una posible adaptación y ampliación del prototipo a un sistema con aplicación real si así fuese necesario. La generación de este prototipo carece de utilidad sin la existencia de un contexto de ejecución que le dote de su propósito, motivo por el cual se idea una funcionalidad transversal, así como un conjunto de escenarios minuciosamente seleccionados, que acompañarán este proyecto. Asimismo se dotará de cuerpo a capacidades como la gestión de tareas, gestión de interrupciones o la provisión de servicios, entre otros, en forma de *kernel* simple, tal que facilitará el estudio de la *MPU*.

Cabría destacar dos cuestiones referidas a la aproximación empleada para el desarrollo. Por un lado, la elección de un lenguaje de programación, tal que este no disponga de mecanismos estáticos para la protección y que a su vez permita el acceso directo a los componentes de la plataforma subyacente sin indireccioneso intervenciones en el control de la operación. Por otro lado, se manifiesta, dadas las particularidades del problema, la permeabilidad de términos del dominio de la solución, esto es aquellos propios de sistema de información, en el dominio del problema, en tanto en cuanto este último forma parte del ámbito de los sistemas empotrados. Del mismo modo, no resulta factible una independencia total –*platform ignorance*– respecto de las arquitecturas de sistemas y sus juegos de instrucciones.

1.2.2 Objetivos

Así pues, el objetivo principal perseguido en este proyecto es la elaboración de un subsistema que habilite la protección entre tareas haciendo uso de la *MPU*. Se busca además lograr cierta independencia, en la medida de lo posible, respecto de los proveedores de las plataformas, lo que permitiría la portabilidad del mecanismo y la reutilización de la lógica propuesta para problemas con similares necesidades de protección. Para ello, será necesario identificar las operaciones primitivas que describen dicho mecanismo, tal que, mediante su modelado y posterior implementación en el lenguaje de programación *C*, sea posible generar un prototipo funcional, el cual evidencie sus bondades frente a situaciones que suponen una amenaza en el entorno de los sistemas empotrados. Dicho producto será integrado en un *kernel* simple, que cumple el papel de soporte en el desarrollo de aplicaciones embebidas.

Se concibe a continuación una lista de objetivos como representación de las meta a alcanzar, mediante el ejercicio aquí recogido:

- Propuesta y elección de la metodología de trabajo empleada para el desarrollo que mejor se ajuste en base a las características del problema.

- Estudio entre distintos mecanismos *hardware* existentes que permiten la protección de memoria, con motivo de indagar cuáles son las condiciones determinantes en la elección de la unidad de protección de memoria frente al resto de técnicas.

- Comparación de distintas realizaciones de *MPU* a nivel comercial, para la posterior extracción de características comunes con independencia de su arquitectura y su posterior clasificación bajo el concepto de operaciones primitivas.

- Concepción de un sistema simple capaz de albergar las operaciones primitivas provistas, a la par que permita la gestión de recursos, así como el tratamiento de excepciones y la planificación de múltiples tareas que constituyen la aplicación embebida.

- Evaluación de las propiedades, eficacia en su labor y alcance del mecanismo propuesto, a través de dos situaciones de aplicación real con potencial riesgo para la ejecución, de forma que se evidencien las bondades y virtudes, así como posibles limitaciones en su uso.

1.3 Estructura del trabajo

La memoria se distribuye en seis capítulos, así como uno adicional bajo el cual se recogen las fuentes bibliográficas y referencias consultadas para la redacción de la misma. Se aporta a continuación un breve sumario relativo a su contenido:

Capítulo 1 – Introducción, que relata la temática central dispuesta para el trabajo, la motivación que da origen a este, el alcance del proyecto, los objetivos fijados y la estructura articulada.

Capítulo 2 – Fundamentos teóricos, en representación de la primera de las actividades a realizar para el proyecto y donde queda reflejada la extracción de conocimiento pergeñado. En su inicio se presenta un estudio de los paradigmas y metodologías que mejor se ajustan al ámbito de los sistemas embebidos, seguido este de la correspondiente elección para cada uno, de modo que se dilucidan las fases genéricas consideradas para el proceso de elaboración *software* y los artefactos generados en cada una. El capítulo avanza presentando las particularidades del contexto tratado, a la par que define un semántica común seguida para la totalidad del trabajo, que en concreto afecta al concepto de estabilidad de la plataforma como temática central para la cual se procura la solución mediante el uso de mecanismos de protección *hadware*. A continuación se someten a evaluación varios de estos mecanismos, con objeto de determinar aquellas causas por las cuales la unidad de protección de memoria resulta una opción optima. Tras ello, se escrutan diversas implementaciones para este módulo, con objeto de identificar aquellas características comunes u operaciones fundamentales presentes en todas ellas. El capítulo concluye con la elección de la arquitectura bajo la cual tendrá lugar la implementación del prototipo.

Capítulo 3 – Obtención de requisitos y análisis, siendo este el capítulo que sienta las bases para el desarrollo, puesto que incorpora en primera instancia una descripción genérica del problema, tal que permite extraer los requerimientos que deberá manejar la solución propuesta para posteriormente conformar los casos de uso, con objeto de recabar las iteracciones existentes entre las entidades y el sistema del que hacen uso. A estos apartados les seguirá un exhaustivo estudio de los conceptos que afectan a áreas propias del dominio del problema, ignorando características relativas a la implementación o la tecnología, de modo que el producto resultante de la actividad pueda considerarse génerico respecto de distintas arquitecturas de sistemas empotrados.

Capítulo 4 – Diseño del sistema, para el que se desarrolla la actividad que estrecha el margen existente entre la concepción de la solución y la tecnología subyacente elegida. En concreto se aborda la lógica *software* a realizar, identificando los componentes que la conforman, su organización, la relación existente entre ellos y su disposición en la plataforma.

Capítulo 5 – Evaluación mediante prueba de concepto, capítulo que recoge dos flujos de simulación, permitiendo así el estudio de las bondades ofrecidas por el mecanismo de protección. La primera se centrará en aquellos aspectos que afectan a la estabilidad del espacio de pila de una tarea, mientras que la segunda responderá a anomalías particulares presentes en el *hardware* empleado.

Capítulo 6 – Conclusiones y líneas futuras, tal que comprende el conjunto de adversidades encontradas en la elaboración del trabajo y para las que se se extraen, reflexión mediante, ideas que permitirán mejorar y depurar las técnicas empleadas de cara a futuros desarrollos. Asimismo se sugieren algunas alternativas posibles para la temática tratada en el trabajo y que permitirían explotar las capacidades que ofrecen las unidades de protección memoria.

Apéndice A – Plan de proyecto, siendo este anexo vital para comprender las actividades aquí desarrolladas. De su actividad resultará la hoja de ruta seguida en este trabajo, indicando los medios necesarios para la elaboración, así como posibles riesgos a los que se deberán hacer frente en cada tarea estimada.

CAPÍTULO 2

FUNDAMENTOS TEÓRICOS

Para la gran mayoría de desarrollos *software* resulta preciso definir un conjunto de procedimientos, con objeto de manejar la complejidad asociada, tal que establecen una hoja de ruta bien definida y cuya puesta en práctica permite el cumplimiento de los objetivos marcados. Los procedimientos empleados, los cuales suelen agruparse en un proceso de desarrollo, varían en función del problema concreto que se pretende resolver, por lo que la existencia de procesos unívocos e infalibles con independencia de la situación se antoja harto imposible. Si bien es cierto, pueden identificarse prácticas comunes aplicables en la mayoría de proyectos, las cuales se documentan en los denominados paradigmas o modelos de desarrollo. Es así que en este capítulo se incluirá la evaluación de una pareja de procesos populares en el ámbito de los sistemas embebidos, de entre los que posteriormente se seleccionará aquel adecuado a las necesidades del proyecto. No obstante, tales modelos no detallan cómo son abordadas las actividades que los conforman, aspecto que será provisto por las denominadas metodologías de desarrollo, hecho que conlleva al análisis de la metodología empleada, así como los motivos que conducen a su elección. [7]

Asimismo, a lo largo de este capítulo se distinguen algunas particularidades en lo que respecta a los sistemas de información empotrados y cómo estos se ven afectados por las condiciones que impone su dominio de aplicación. De igual modo se pretende establecer una distinción semántica entre los términos seguridad y estabilidad de un sistema de información, incidiendo, en lo que respecta a este trabajo, en este segundo, así como también se fijará el significado del término protección de memoria y su interpretación bajo este tipo de sistemas.

Por último, se exponen algunas de los distintas concepciones de mecanismos *hardware* de protección de memoria actualmente existentes, las características que los definen y las ventajas o desventajas que presentan unos frente a otros, permitiendo así justificar la elección de la *MPU*. Tras ello, se procede a la comparativa, mediante el uso de parámetros de interés, de implementaciones comerciales existentes para este mecanismo, de entre las cuales se obtiene aquella seleccionada como candidata, no sin antes identificar algunas operaciones comunes para todas y cada una de ellas, que serán aquellas a incorporar en el prototipo.

2.1 PARADIGMA Y METODOLOGÍA DE DESARROLLO EMPLEADOS

Los denominados modelos de desarrollo software, pese a definir procesos de elaboración de forma abstracta y sin necesidad de conocer detalles específicos, sí concretan cómo abordar la organización de dichas actividades, de manera que cada uno ofrece diferentes ventajas respecto al resto en función del ámbito de aplicación. En lo que se refiere a este trabajo, se imponen dos restricciones que repercuten a dicha organización y afectan por tanto a la decisión del modelo escogido –el paradigma escogido deberá ajustarse a la naturaleza de estos–, a saber: la elaboración de las tareas contempladas tiene lugar en exclusiva por una persona, por lo que no se requieren procesos para su sincronización y gestión, y el ámbito de aplicación y

estudio para este caso es el de los sistemas empotrados. Las características particulares que conforman a estos sistemas, serán detalladas en lo sucesivo, pero cabe destacar por el momento, la variedad de restricciones que su entorno impone sobre ellos, en cuanto que presentan riesgos extrínsecos, que afectan a su funcionamiento, así como intrínsecos, los cuales pueden amenazar la seguridad de las entidades que utilizan directa o indirectamente estos sistemas. Por ello, se demanda un paradigma de desarrollo fuertemente estructurado, de manera que se dispone en caso de detección de fallo durante el proceso de elaboración de un punto estable para retorno, hecho que requiere de la captura de la información que generan las actividades desarrolladas como condición *sine qua non*, logrando así constancia de todas y cada una de las decisiones planteadas. Además, se plantea necesario el minucioso análisis de las características y limitaciones –conocidas en gran medida *a priori*, aunque cabe la posibilidad de aparición a lo largo del desarrollo– que recoge el dominio del problema, las cuales se documentan obligatoriamente bajo una lista de requisitos bien definida, cuyo estado deberá permanecer inmutable como soporte en el desarrollo. De entre las posibles opciones estudiadas y dadas las consideraciones anteriores, únicamente se admiten dos propuestas, que en parte se encuentran estrechamente relacionadas.

A continuación se presentan dichas opciones, su descripción de manera resumida y los puntos de convergencia existentes entre estas:

- El primero de los paradigmas aquí planteados es el modelo en espiral, propuesto originalmente por *Barry W. Boehm* en [8] y [9]. Sumariamente, se define como la ejecución del desarrollo *software* siguiendo un patrón incremental orientado a riesgos, donde el orden y la secuencia lógica de las actividades se determina de manera dinámica –aunque no normativa–, en virtud de las amenazas existentes y su magnitud. El principal beneficio obtenido es la identificación de problemas presentes en fases tempranas del desarrollo, pudiendo así establecer los mecanismos adecuados para su rectificación, lo que que permite hacer frente a cambios radicales fruto de nuevas especificaciones de cliente o en condición de errores asociados al propio proceso. Se considera un modelo un tanto especial, en cuanto que aglutina diversas técnicas, a la vez que su núcleo se utiliza para la definición de otros paradigmas e incluso metodologías, tales como *RUP* [10] [11] y *MBASE* [12], siendo esta última de gran interés. El proceso que describe es replicado de manera cíclica, donde cada repetición, también identificada bajo el concepto de ronda, permite el progreso en el desarrollo y donde tales repeticiones son caracterizadas visualmente como una espiral, tal que el eje radial representa el coste acumulado, mientras que la dimensión angular refiere el avance realizado –las rondas interiores corresponden al inicio del desarrollo y la exteriores muestran el avance de la solución–. Existen cuatro actividades fundamentales para cada ronda, las cuales son: determinación de objetivos y restricciones para la porción del producto a elaborar, evaluación e identificación de los riesgos presentes –riesgo como posible causa de fallo del proyecto–, desarrollo y verificación del trabajo realizado y planificación de sucesivas fases. Como resultado se obtienen artefactos *software*, que involucran a ambas partes (cliente y desarrolladores), de manera que las decisiones que estos toman, definen en lo sucesivo el comportamiento de la aplicación. No obstante, una naturaleza cambiante para el problema que se resuelve, se contrapone a las necesidades de los sistemas empotrados, que necesitan un punto de partida definido e invariable para su realización, lo que descarta a este paradigma para los intereses de este trabajo.

- El segundo paradigma evaluado es el comúnmente denominado como modelo en cascada, el cual se concibe como uno de los paradigmas orientados a la planificación, tal que se establece una serie de actividades secuenciales a desarrollar, así como el orden concreto para cada una de ellas al inicio del proyecto y del mismo modo, se acuerdan los requerimientos de la aplicación en los primeros compases, lo que impide cualquier posibilidad de modificación. El resultado para cada actividad, se expresa

como un artefacto el cual apenas sufre alteraciones a lo largo del proceso de elaboración y que a diferencia de aproximaciones iterativas incrementales, carece de utilidad para los clientes. Esta definición, aunque frecuente es en realidad errónea, en tanto que se malinterpreta –tal y como apunta [13], que recoge algunas de las discrepancias– la descripción original recogida en [14], por lo que se considera necesario realizar una breve aclaración al respecto, debido a que en la actualidad, este paradigma se percibe como obsoleto para cualquier planificación de proyecto y desarrollo *software*. Si bien es cierto, la falta de una definición cerrada y contundente desde el principio abre la posibilidad a estas interpretaciones, pero bajo una exhaustiva lectura, se identifican elementos presentes en modelos propiamente iterativos e incrementales, que lo alejan del carácter secuencial tan ampliamente extendido. Dadas pues las circunstancias mencionadas, se referirá, en lo que compete a este trabajo, al artículo original a la hora de establecer las actividades y características para el modelo y no así a las revisiones posteriores. Cabe aclarar que no se impone límite al número de veces que se procede bajo una determinada actividad y en la misma medida su resultado –entre los que se incluyen los requisitos– no permanece estático, en cuanto que se prevén cambios para fases anteriores –lo que implica su ejecución–, generalmente motivados tras las fases de prueba y evaluación del producto elaborado, de manera que no es correcto considerar un carácter inmutable para el modelo. Con objeto de manejar la complejidad asociada a la aparición de dichas modificaciones, el paradigma plantea diversos mecanismos, entre los que destacan:

- Fuerte uso de documentación, siendo este uno de los artefactos esperados tras completar una actividad y con la particularidad de actuar como especificación y diseño del sistema en fases iniciales.
- Elaboración de un diseño preliminar previo a cualquier consideración, tal que permita identificar, bajo los primeros instantes del desarrollo, problemas subyacentes asociados a fases posteriores.
- Implementación de una versión piloto –o prototipo si se quiere– paralela a la elaboración principal sistema, por lo que se rechaza la idea de carácter secuencial, de forma que se logra involucrar al cliente en el procedimiento, hecho que demuestra que el resultado de la actividad aporta valor, siendo este un punto de convergencia respecto del paradigma en espiral.

Dada la finalidad de este proyecto, bajo el que se pretende elaborar un prototipo que facilite el estudio y refleje el comportamiento de los mecanismos de protección y no así un sistema operacional completo, el modelo que mejor se ajusta y por tanto aquel utilizado, es el paradigma en cascada –se hace hincapié en que la diferencia para la definición expuesta del término–. Sin embargo, dada la existencia de puntos de unión entre ambos modelos presentados, se plantea la utilización del paradigma en espiral –dada la facilidad de integración que presenta– como futura posibilidad, por ejemplo, en ampliación de la funcionalidad del prototipo. Es preciso apuntar la existencia de derivados para el paradigma elegido, tal que soportan la formalización, es decir, el empleo de modelos matemáticos con el fin de validar el grado de corrección entre los requisitos especificados y la implementación, de manera que su aplicación es extremadamente atractiva para los sistemas empotrados, con estrictas necesidades en materia de seguridad y estabilidad para la ejecución.

Una vez queda definido el paradigma a emplear, se requiere concretar la realización de las actividades identificadas, mediante la correspondiente metodología de desarrollo. Puesta de manifiesta la importancia sobre la documentación para este tipo de sistemas y el uso que hace de esta el paradigma utilizado, es conveniente dilucidar la información a reunir y la manera más adecuada para su representación. La documentación deberá incorporar aquellos aspectos que describan el contexto del dominio, con objeto de comprender el entorno que envuelve al problema para el que se ofrece solución –identificando así las propiedades antes de la elaboración del producto mismo– que por otro lado, detallará los mecanismos y aproximaciones planteadas

como solución, tal que existe una definición precisa en el modo de obrar, que sirve como punto de referencia para el desarrollo y establece un lenguaje común para desarrolladores y clientes. La metodología elegida, por tanto, deberá facilitar la producción de dicha documentación y en la misma medida, su tarea requiere abordar la complejidad que supone la aparición de cambios, fruto de las actividades, que afectan a las especificaciones iniciales, al diseño y al producto desarrollado. Por ende, se aislarán las modificaciones surgidas a un ámbito controlado, a la par que se mitigan los costes en tiempo y recursos que suponen al proyecto tales cambios, conservando en la medida de lo posible el trabajo previamente desarrollado y previniendo la introducción de errores en el proceso.

La propuesta de metodología considerada para este trabajo, corresponde a aquellas orientadas a modelos –donde en lo sucesivo, el uso del término modelo ya no refiere a un sustitutivo de paradigma– como productos resultantes de una actividad, donde el campo que abarca tales metodologías se conoce como *Model Driven Engineering*. Los modelos empleados incorporan sólo aquellas propiedades, asociadas tanto a entidades del dominio como a operaciones del sistema, necesarias para su representación en términos del proceso de elaboración o dada una perspectiva concreta, eliminando la complejidad asociada a su definición real. Además, permiten su formalización, de forma que se evalúa el grado de adecuación entre la entidad real respecto de su representación, tal que es factible concretar la mejor solución previo paso a la implementación, que se ajusta dado unos parámetros concretos. Resulta de interés la capacidad de los modelos para generar valor como vehículos en la comunicación de información, así como su utilidad en la síntesis de artefactos *software* a partir de ellos o en la representación del *hardware* de la plataforma, siendo este último efecto visible, en cuanto que disminuye los costes asociados en el desarrollo. Es tipo de metodologías define un espacio de modelado, esto es, el conjunto de modelos que representan el sistema, tal que pueden clasificarse bien por el área representan –por ejemplo, el área de negocio– o por su nivel de abstracción, es decir, la proximidad respecto al producto final. Aquella escogida para el trabajo, se conoce como *Model Driven Architecture* [15] –*MDA*® creado por el *Object Management Group* [16] como consortio para la generación de estándares– la cual aborda aspectos concretos para el análisis, desarrollo e implementación del producto *software*.

2.1.1 MDA®

MDA® se define pues como un procedimiento de desarrollo *software* orientado a modelo, que proporciona una serie de pautas precisas, cuya aplicación permite desacoplar la lógica de la aplicación de los requerimientos impuestos por la infraestructura subyacente, de peso para la implementación, facilitando así la adaptación en cambio de tecnologías. Para cada operación, se obtienen modelos –entidades de primer orden, cuya representación no se limita de manera exclusiva al uso de diagramas, sino que se contemplan numerosos formatos– como productos, donde cada uno de ellos se incluye en un estadio *n*, el cual se interpreta como un nivel de abstracción, conformado por uno o múltiples modelos. Cada nivel de la metodología, se coordina bajo un lenguaje de modelado común, que ofrece la semántica y notación utilizada, y que delimita las características y el comportamiento de los modelos con los que se trabaja, cohesionando los elementos que los conforman. Puesto que todo artefacto se considera un modelo –entidades de primer orden–, se produce un fenómeno de circularidad, de forma que el origen del lenguaje de modelado empleado se realiza a través de otro modelo, conocido este como metamodelo y situado un en el nivel de abstracción inmediatamente superior. En el nivel más alto de la abstracción, se establece un límite del cual se derivan el resto de características para dichos lenguajes, siendo *Meta Object Facility* [17] (*MOF*™), el cual define una serie de capas o niveles para el metamodelado, donde el propio modelo de *MOF*™se sitúa en el nivel más alto, esto es *M3* — la elección considerada con este fin. Este provee de un conjunto de propiedades para la definición de los componentes que actúan en los modelos (metamodelos y metametamodelos), así

como un lenguaje común al que se adhieren los niveles inferiores. Mientras, el lenguaje de modelado escogido para la representación de la solución y situado en el nivel *M1* de *MOF*™, se conoce como UML® [18]. [19]

El procedimiento clasifica los distintos modelos respecto de su nivel de abstracción, tomando como punto de referencia la arquitectura, no en un sentido tecnológico, sino como el conjunto de modelos que representan el sistema a elaborar, así como la actividad que los lleva a cabo. Aunque no se define el nivel de profundidad para la abstracción, sí facilita una clasificación de modelos a grandes rasgos, respecto de su posición –de más a menos abstracción–, los cuales se recogen brevemente a continuación –descritas en [19]–:

- Modelos de dominio, los cuales aglutinan y presentan información relativa al problema para el que se ofrece solución, la cual puede referirse a diversas temáticas, entre las que se incluyen los sistemas de computación.
- Modelos de lógica de sistema, que reflejan cómo interactúan los elementos que conforman la solución entre ellos y cómo permiten a los usuarios alcanzar sus metas.
- Modelos de implementación, los cuales describen la forma concreta de abordar la realización para el sistema solución planteado y por tanto dependientes de la tecnología subyacente.

De nuevo y respecto de la arquitectura, se plantean dos conceptos, tal que reflejan los intereses sobre una parte concreta del sistema, así como la representación particular que se ajusta a dichos intereses –sin incorporar detalles propios de la implementación–, con motivo de orientar su concepción, los cuales reciben el nombre de punto de vista (*viewpoint*) y perspectiva *view*) respectivamente. Así por ejemplo, si se supone un sistema de envío de mensajes, el intercambio de información se consideraría el punto de vista, mientras la perspectiva plantea cómo se estructura la información que resulta compartida. En lo que se refiere a este trabajo, el punto de vista se limitaría a la estabilidad de los sistemas de información tratados –en concreto a la seguridad de las operaciones que involucran a la memoria– y la perspectiva indicaría aspectos concretos, por ejemplo, qué técnicas permiten controlar los accesos. Cabe destacar la conflictividad para el término punto de vista bajo este trabajo, en cuanto que su significado difiere para el momento de aplicación, de manera que referirá a la definición descrita en este párrafo exclusivamente para la extensión de este capítulo.

Puesto que un modelo representa la evolución del sistema para un momento determinado del desarrollo, aquellos que corresponden a los niveles más abstractos, generalmente representan al sistema en un estado embrionario y a medida que continua el desarrollo, los modelos resultantes se aproximan a los artefactos *software*, que constituyen el resultado final. Entre los diferentes modelos, cada uno en representación de un nivel de abstracción, existe una correspondencia en forma de continuidad, de manera que la síntesis de un modelo, la cual mayoritariamente tiene como origen un nivel de abstracción diferente al de destino, es posible a través del resto y en primera instancia, es decir en ausencia de modelos, desde los requisitos establecidos. Esta técnica, que se conoce como transformación, es la característica más representativa para *MDA*, en cuanto que permite un desarrollo iterativo con cambios incrementales, tal que se conserva el trabajo previamente realizado –efecto además compatible con la visión del paradigma utilizado en este trabajo–. El sentido de la transformación se permite bien de manera descendente respecto de la abstracción, estos es, hacia las particularidades de la infraestructura subyacente, en donde por ejemplo se encuentran los artefactos *software* –considerados como modelos para algunos autores– o por el contrario ascendentemente, aunque esta ultima posibilidad se plantea en ocasiones difícil de manejar, dado que requiere incorporar modificaciones a modelos ya constituidos. Gracias a este mecanismo, se logra mitigar el efecto que produce el fenómeno conocido como decaimiento arquitectónico [20], donde cambios producidos por la evolución en el desarrollo no se reflejan o tipifican en las especificaciones de la arquitectura inicial. Existen dos vías para

la transformación [21], donde la primera recibe el nombre de elaboración, en referencia al carácter manual para el proceso, en cuanto que los modelos tan sólo recogen la información necesaria a representar y esta se incorpora de forma mecánica por el desarrollador. La segunda, la cual se conoce como traducción, considera a los modelos como una soluciones completas, esto es, albergan la información correspondiente a su nivel de abstracción, así como aquella que facilita la tarea de conversión, la cual tiene lugar de manera automática por los denominados compiladores de modelos. Ambas aproximaciones pueden coexistir para el mismo desarrollo, tal que por ejemplo, durante el inicio se establece una transformación de carácter manual y llegado el momento, se orienta a aquella que incluye transformaciones automáticas.

El grado de corrección, dada una traducción, se determina por las denominadas reglas de transformación, tal que describen como los elementos –los cuales constituyen el modelo de entrada–, se corresponden y adecúan a los atributos del modelo resultante, donde además se especifican qué extensiones respecto del modelo original se deben incorporar si fuera necesario. Tanto el grado y dirección para la correspondencia entre modelos se representan para MDA bajo lenguajes exclusivos para este fin, los cuales se adhieren a una especificación común denominada como *Query/View/Transformation* [22] (*QVT*™), de forma que es posible establecer un metamodelo para las reglas de transformación, que a su vez incorpora e interpreta semántica propia de los metamodelos que definen a aquellos implicados en la conversión, esto es, origen y destino. El resultado de la operación es verificable mediante la depuración de una traza de transformación, como histórico para la ejecución del proceso. Cabe mencionar que la generación de código a partir de un modelo, tan sólo es posible siempre y cuando dicho modelo sea ejecutable, esto es, sólo cuando se definen bajo este todas y cada una de las posibles operaciones que debe realizar el sistema representado.

MDA® distingue al menos cuatro tipos de modelo respecto de su nivel de abstracción:

- Modelo independiente de aspectos computacionales (*CIM*), tal y como su propio nombre indica, es ajeno a cualquier aspecto relacionado con la lógica de la solución, de manera que no incorpora ningún detalle acerca de la construcción del sistema previsto, lo que lo sitúa en el nivel más alto de la abstracción. Su tarea se limita a la representación de la ontología para el dominio del problema, es decir, establece el vocabulario y la semántica para un contexto de aplicación determinado, así como los procesos de negocio, los requisitos y requerimientos, como punto de partida del desarrollo, que deberá incorporar la aplicación. Este modelo admite diversos formatos de diagramas para su representaciones, entre los que se encuentran: *BPMN*, *DFD* –foco en el paso de información–, diagrama de actividad –que refleja las actividades propias del dominio y las condiciones para el flujo– y por último el modelo de características, el cual se evalúa en el trabajo [23]. En ocasiones es frecuente el uso de diagramas de caso de uso en la representación, pero esto resulta incorrecto, dado que estos incorporan operaciones relativas a los sistemas de información, tal que ahondan en la respuesta del sistema frente una acción del usuario, función que no compete al *CIM*.
- Modelo independiente de la plataforma (*PIM*), siendo esta quizás la especificación de modelo más interesante para la metodología dada sus connotaciones y para la que es preciso realizar una breve aclaración al respecto. Dado el nivel de abstracción al que pertenece, un modelo puede ser ajeno a determinados aspectos para la tecnología subyacente, lo que le confiere una cualidad de cierta independencia. No obstante, la identificación de un modelo como *PIM* es relativa, siempre respecto al resto de modelos, esto es, aquellos que lo preceden y suceden, y por tanto condicionada por su momento de realización en el desarrollo. Por tanto, un modelo es caracterizado como *PIM* respecto de aquel que le sigue, mientras que actúa como *PSM* para el anterior, en cuanto a su proximidad frente a la infraestructura bajo la que se aloja el sistema. Este tipo de modelos describen la funcionalidad y el comportamiento para solución propuesta, priorizando la portabilidad entre plataformas, de manera

que se identifican las operaciones que responden a las demandas de aquellas entidades, las cuales interaccionan con la aplicación. Destaca la dificultad en su creación a partir de modelos *CIM* –reflejado en [24]–, hecho que puede suponer un cuello de botella para el desarrollo, dada la complejidad de traducción para este proceso, por lo que se demanda la intervención manual de un experto del dominio, capaz de convertir la semántica propia del problema en actividades del sistema. Es habitual su representación como diagramas de componentes –eliminando los componentes físicos si fuera necesario y haciendo énfasis en los componentes lógicos–, de caso de uso, de secuencia, de clase, de paquetes y máquinas de estado.

- Modelo específico de la plataforma (*PSM*), el cual arroja mayor detalle respecto de las características de la plataforma, frente al modelo presente en el nivel de abstracción que lo antecede, con el que comparte mecanismos para la representación. Por ello, la transformación desde modelos anteriores es trivial y del mismo modo, se permiten transformaciones en sentido inverso, con objeto mantener la coherencia para la representación –incluyendo cambios propios del avance en el desarrollo–.

- Modelo específico de la implementación (*ISM*), donde en concreto no se contempla con esta terminología bajo la definición de *OMG*®, sino que aparece trabajos como [25]. Se considera conveniente su inclusión, dada su particularidad, tal que se refiere al propia código de la aplicación. Este modelo –por ende la implementación del sistema– se deriva del *PSM*, incorporando detalles asociados a la plataforma. Además, si el desarrollo lo permite, este modelo puede reemplazar al *PSM* y derivarse directamente desde el *PIM*.

Para este trabajo, los modelos cumplirán en un principio el papel de documentación para el desarrollo realizado, lo que implica una correspondencia 1:1 entre el modelo y los artefactos *software*, pero no se descarta dadas las características de la metodología propuesta, su empleo en la generación de ejecutables. Por ello, las transformaciones aquí empleadas presentan un cariz manual, tal que no se establece un metamodelo para la traducción, pero se contempla un traslado a la generación automática como futurible, en cuanto la producción de sistemas se vería beneficiada –en eficiencia, reducción de errores al incorporar actualizaciones, así como coherencia entre dichas modificaciones–. Se demanda así el empleo de una herramienta *CASE* (*Computer-aided Software Engineering*), que soporte la metodología *MDA*® y que siga las líneas de estandarización planteadas por *OMG*® para la creación de metamodelos y esquemas de traducción. *Papyrus*™ [26], el cual pertenece al proyecto de herramientas para desarrollo de modelos (*MDT*) bajo el *IDE Eclipse* [27], resulta la elección *software* fruto de esta condición, tal que proporciona en forma de interfaz gráfica los medios necesarios para la creación de modelos, con soporte para la especificación *UML*®–por extensión *MOF*™–, así como reglas basadas en *QVT*™, habilitando la transformación de modelos *MOF*™a código de aplicación –para lenguajes de programación como *C++* y *C*–. Sin embargo, limitaciones de uso presentes para versiones actuales de la herramienta, las cuales afectan a la elaboración de diagramas de secuencia, obligan a la elección de *PlantUML* [28] como sustitutivo para completar la tarea. Al igual que *Papyrus*, esta herramienta permite su integración en el entorno de desarrollo *Eclipse*, por lo que los modelos que esta genera permanecen bajo el mismo proyecto, facilitando así su exportación si se requiere de nuevo el empleo de *Papyrus*, la cual tendrá lugar en última instancia de forma manual –aunque sólo es necesaria una iteración para este proceso–, en cuanto que *PlantUML* no soporta la transformación de modelos, así como el formato empleado por *Papyrus* para la representación de diagramas. *PlantUML* destaca por su rapidez en la generación de modelos, ya que produce elementos gráficos como traducción automática de representaciones en texto plano, ofreciendo una excelente alternativa para el desarrollo de prototipos, donde la inversión en recursos se establece en el contenido de dichos modelos y no así en la curva de aprendizaje.

2.2 Consideraciones sobre los sistemas empotrados

Sin lugar a dudas, dentro de los sistemas de información, los denominados sistemas de cómputo general –entre los que se encuentran ordenadores personales o teléfonos inteligentes– resultan los dispositivos de consumo con mayor disposición –al menos de forma aparente– en la actualidad. Si bien, este término –sistema de propósito general– cobra diferentes sentidos, hasta el punto de perder su concepción inicial y con esta su capacidad para clasificar conjuntos de dispositivos de computación para unas características dadas, motivo por el cual se plantea la siguiente aproximación para el trabajo:

> Se trata de aquellos sistemas para los que su infraestructura y funcionalidad no se encuentran acotadas, es decir, a lo largo de su tiempo de vida realizan tareas dispares, las cuales no son predefinidas en su concepción ni tampoco guardan relación entre sí. Además, presentan mecanismos específicos que facilitan la labor de adaptación a las necesidades de las entidades que interactúan con ellos.

A pesar de contar con una inadvertida presencia –puesto que su volumen comercial es mayor si cabe–, junto a los sistemas de cómputo general –aunque también de manera independiente– se encuentran los sistemas empotrados, los cuales generalmente no son accesibles ni admiten una manipulación directa, por lo que su función remite a aquella de soporte para los sistemas bajo los que se agregan –por ejemplo, en la captura de información del entorno mediante la manipulación de un sensor–, donde asimismo garantizan su correcta ejecución. Puesto que en lo sucesivo se establece el foco en este campo, resulta indispensable establecer una descripción concisa, tal que incorpore aquellas características más destacables para los sistemas embebidos, en cuanto que al igual que ocurre para aquellos de cómputo general, su semántica es dispar, dadas las variaciones sufridas respecto de su concepción inicial.

Este tipo de sistemas se configuran mediante tres pilares fundamentales, a saber:

- El entorno o dominio de aplicación, es decir, el ámbito concreto donde los sistemas empotrados desempeñan su función. Este aspecto determina la complejidad del sistema y por ende las medidas y consideraciones a tomar durante su desarrollo –el artículo [29] presenta un interesante resumen acerca de las condiciones que impone dicho entorno–.

- La infraestructura o *hardware* subyacente. A menudo, términos como *microcontrolador*, microprocesador o microcomputador se utilizan de forma unívoca para identificar al circuito integrado y la unidad central de procesamiento, las cuales ofrecen soporte para la ejecución de lógica. La labor de identificar las diferencias –si existen– entre estos excede las competencias del trabajo, aunque sin embargo, sí se pretende establecer el *modus operandi* escogido a la hora de relacionar estos términos para la identificación de la plataforma hardware. La aplicación de estos términos tanto de manera histórica [30] [31] como actual [32] [33] [34] –por tanto aquella empleada en este documento– obedece a la segmentación sobre el catálogo de producto que ofrecen los fabricantes, esto es, en base por un lado, al rendimiento en computación y la inclusión de un mecanismo para la traducción de direcciones lógicas, tal que dicha descripción corresponde a aquellos referidos como sistemas conformados por microprocesador –que dadas sus capacidades no requieren de un propósito específico–, mientras que por otro, *microcontrolador* se asocia a una limitación en recursos sin soporte de traducción, indicados para una actividad concreta. No obstante, es frecuente ver el uso indiscriminado de ambos términos, de forma que su definición se aproxima hasta confundirse.

- La aplicación contenida, la cual tiene lugar bajo el sistema determinado por los puntos anteriores. En ocasiones, partes de su lógica –siempre y cuando el impacto en rendimiento sea despreciable– actúan como sustitutivos de elementos de la plataforma subyacente, de forma se reducen costes recurrente

asociados al dispositivo elegido para la realización del proyecto. En el caso antitético, su funcionalidad se reemplaza por un componente *hardware*, en cuyo caso aún sería necesario proporcionar igualmente el código de interacción (*driver*) para el sistema.

Tal y como se ha mencionado, un sistema embebido se integra de forma física en otros sistemas, que no necesariamente deben ser computacionales –por ejemplo, el sistema de control de transmisión de un coche, donde las partes que conforman el sistema operacional son mecánicas–, de forma que su domino de aplicación siempre es especificado. La labor de los sistemas empotrados es el control –puede requerir el procesado de información– de aquel en el que se incluye, garantizando así el correcto funcionamiento de sus partes. Al ser contenido bajo un ámbito de ejecución, el entorno impone una serie de restricciones tanto *software* como *hardware*, las cuales varían entre dominios, por lo que cada desarrollo presenta un desafío –aunque existen metodologías que permiten el uso y reutilización de módulos–.

Como refleja [35], algunas de estas restricciones abarcan desde el coste máximo permitido para el proyecto –que determina los componentes, normalmente integrados bajo un mismo paquete, a utilizar–, la tasa de transferencia de información –definida por el dominio– que debe procesar por unidad de tiempo o el determinismo –asociado a la estimación del peor tiempo en ejecución del flujo, así como a la previsibilidad del comportamiento para el sistema–. Asimismo, en función de la complejidad de los entornos bajo los que se integra, por ejemplo, el sistema no es accesible físicamente, de manera que no se puede reparar un elemento inutilizado, también se imponen limitaciones tales como la fiabilidad –que estudia la probabilidad de que el sistema proporcione el funcionamiento correcto de forma consistente–, la robustez o estabilidad –como la habilidad del sistema para proporcionar sus servicios aún cuando se incumplen las condiciones normales de operación– y por último la seguridad –que indica el nivel de riesgo en el uso del sistema, tal que el resultado de su acción desemboca en la pérdida de datos o en accidente–. Cabe destacar que el sentido de los términos estabilidad y seguridad (tomados de [35]) se cuestiona en el siguiente apartado, de modo que se proporciona una definición, la cual se considera más acertada, hecho que tiene como resultado el intercambio de su semántica. Con motivo de facilitar el cumplimiento de los mencionados requerimientos, algunas plataformas proveen de mecanismos tales como *Power-On Self-Test*(*POST*), *Built-in Self-Test* (*BIST*) o la comprobación de permisos mediante el control de operaciones sobre memoria, donde este último presenta un gran interés en lo que a este documento se refiere.

2.3 Fundamentos de la protección de memoria

2.3.1 Seguridad y estabilidad del sistema

De igual forma que se evidencia en el anterior apartado, algunas de las limitaciones a las que el desarrollo de los sistemas empotrados deberá hacer frente refieren a la propiedad de protección, donde se distingue tanto la protección frente a agentes externos o por el contrario, la protección hacia su entorno. Ambas ideas de protección se encuentran estrechamente relacionadas, puesto que si se manipula de manera extrínseca la plataforma, dicha acción acaba repercutiendo en su dominio de aplicación,y de manera circular, un cambio en el entorno –provocado de forma inicial por condiciones ajenas al propio sistema– se propaga a las características intrínsecas del sistema. Para su designación, se utilizan términos tales como estabilidad y seguridad, los cuales generalmente suelen confundirse entre sí, sobre todo bajo los manuales de referencia de fabricantes, así como en la literatura que trata tales cuestiones en el contexto de los sistemas empotrados. Por ello y dado el contexto de este documento, se establecen una serie de descripciones, que actúan como coordenadas, con objeto de referenciar tales ideas sin confusión alguna.

El artículo [36] introduce dos términos –propios del habla inglesa–, que pretenden establecer de forma complementaria la relación existente entre un sistema y su ámbito de operación, y viceversa. El primer término se conoce como seguridad (*security*) y hace referencia a la imposibilidad del sistema para ser afectado negativamente por su marco de ejecución, garantizando además una serie de atributos tales como confidencialidad, integridad o disponibilidad. Esta cualidad asegura que ningún ente malicioso externo pueda ocasionar perjuicios a la plataforma, hecho que conllevaría a un flujo incorrecto de ejecución, generando daños en su entorno de operación –completando así una circularidad, la cual entronca con el siguiente término–. Esta definición concuerda con [37], bajo el cual además se presenta el uso de mecanismo de protección en la línea de este documento. Con respecto a la segunda expresión (*safety*), se convierte en una problemática encontrar aquella unidad lingüística del español, la cual recoge su semántica. Generalmente este término indica la cualidad por la que un sistema no influye perjudicialmente en su entorno, esto es, que no implique daños irreversibles –pérdida de vidas humanas, desastres medioambientales o sencillamente la destrucción de la infraestructura en la cual se integra el sistema empotrado–. Con objeto de garantizar dicha cualidad, el sistema se diseña contemplando los potenciales riesgos fruto de su acción, a la par que se establece tanto un estado conocido y determinado de ejecución, así como los mecanismos que permitan regresar a dicho estado o ante la negativa, impedir un flujo anormal y no previsto durante la ejecución. El término en español, que mejor precisa –y será empleado a lo largo del documento– esta característica, es el de estabilidad del sistema, entendido como la imposibilidad de cambio y la capacidad de recuperación de equilibrio, tal que equilibrio representa el estado de ejecución confiable, que no supone un riesgo para su campo de aplicación.

La atención bajo este trabajo se volcará pues, en aquellos mecanismos que las plataformas ofrecen para garantizar el flujo correcto de ejecución, esto es, su estabilidad. De entre dichas técnicas de protección, es posible encontrar la protección de memoria, cuyo significado se tratará en el siguiente apartado.

2.3.2 Protección de memoria

En el conjunto de arquitecturas de sistemas de computadores actualmente existentes, los distintos módulos lógicos –entre los que se encuentra la memoria principal del sistema, sin la cual no es posible la operación de la aplicación–, que componen la plataforma y ofrecen servicio a las aplicaciones del sistema, son alojados en el denominado espacio de direcciones. Si se desea por tanto acceder a un recurso, será necesario proporcionar una dirección, la cual se define como dirección física –haciendo referencia a la plataforma *hardware* subyacente– e identifica la entrada para la decodificación en el bus de direcciones del sistema. Cada recurso del sistema dispone en este espacio de una serie de registros de control y configuración, así como en ocasiones de un espacio para datos.

Bajo los sistemas de información es común la presencia de varios procesos, que constituyen la aplicación y cuyo ciclo de vida coincide en el mismo intervalo de tiempo, a los que denominamos procesos concurrentes, tal que operan bajo el mismo espacio de direcciones, por lo que los recursos disponibles en la infraestructura son compartidos. Asimismo, un proceso puede disponer del conjunto de datos e instrucciones que constituyen a otro proceso –en forma de espacio compartido de memoria y librerías–, con el objetivo de interactuar con este o modificar su comportamiento. Dada una situación para la que no se limita el tiempo de ejecución, el número de recursos accesibles, ni así como los permisos en operación para una tarea, y en ausencia de un mecanismo que permita su aislamiento, se produce un fenómeno bajo el que un proceso consigue disponer a su antojo e indefinidamente del conjunto de la plataforma, negando a otros el acceso. Las consecuencias de este hecho comprenden desde el bloqueo parcial o total del sistema, la corrupción del contenido de la memoria y la pérdida irreversible de datos, y en definitiva, un estado no operativo del sistema –si la tarea no opera bajo los términos con los que fue establecida por manipulación directa o indirecta– que conlleva a

la incapacidad de completar los objetivos para los cuales fue diseñado. Por ello, dada una situación como la anterior propuesta y en lo relativo a dichos recursos, se presta de vital importancia consolidar su correcto uso, de modo que ningún proceso acapare o prive de ejecución al resto, y en concreto en lo que respecta al acceso y manipulación de la memoria principal, será preciso asegurar la confiabilidad y disponibilidad, esto es, una tarea deberá poder realizar todas las operaciones sobre la memoria que se conciben para ella o ante la negativa, el sistema indicará las causas –previamente estipuladas– por las que se deniega cualquier acción destinada a su uso. Surge así el incentivo de presentar una técnica, la cual se conoce como protección de memoria, que impida a un proceso capturar toda la lógica del sistema, modificar e interactuar con el espacio asignado a otros procesos y en definitiva garantizar que no se exceden las capacidades para las cuales fue programado. Este mecanismo destaca por su capacidad para evitar accesos indebidos –no previsto en el flujo de la aplicación– a una posición del espacio de direcciones física.

Un proceso se considera de confianza por el sistema, si se conocen tanto su origen como su lógica, –es decir, están previstas las operaciones que debe realizar–, mientras que se identifica como no confiable en caso de ignorar su origen o representación interna. Ambas categorías de procesos pueden realizar operaciones que suponen una amenaza –bien de forma deliberada o no– para la integridad del sistema y en consecuencia para su estabilidad, aunque si bien es cierto, la utilidad que presenta la protección de memoria para cada una de ellas se plantea desde distintas coordenadas. Las características físicas del entorno –radiación electromagnética o propiedades eléctricas de la plataforma, como ejemplo de circularidad entre la influencia que tiene el entorno sobre el sistema y vuelta– o características intrínsecas a la aplicación –como errores en la codificación–, provocan alteraciones en la lógica o en los datos de un proceso, que tiene como resultando un comportamiento no previsto. Para estos casos, la protección de memoria asevera que los cambios producidos en la aleatoriedad del comportamiento no afectan a la totalidad del sistema –por ejemplo, modificando áreas en propiedad de otro proceso– o si esto no es posible, deberá revertir o ignorar tales modificaciones, con objeto de presentar un estado conocido, determinado y operativo de este. A mayores, el mecanismo deberá afirmar respecto de los procesos confiables dos características en su ejecución, precisadas como *memory-safety* y *type-safety* respectivamente [38]. La primera se especifica como la validez de las referencias a memoria, esto es, los enlaces a las entidades que residen en ella, comprobando que la posición a la que apuntan se incluye en el espacio de direcciones implementado para la plataforma y asociado al proceso interesado. El segundo término se detalla como la garantía en la corrección de las operaciones, es decir, estas son sólo aquellas establecidas para la arquitectura que constituye la plataforma, con privilegio el privilegio de acceso indicado y para el tipo definido del objeto accedido. Para aquellos procesos considerados hostiles –no es posible determinar si han sido modificados o su representación no corresponde a los establecidos–, el mecanismo de protección deberá asegurar que el resultado de su ejecución no produce cambio alguno para cualquier punto del sistema –exceptuando los espacios asignados a dicho proceso–. Dada la gravedad de su amenaza, en ocasiones es preciso aislar este tipo del resto de procesos y recursos, bajo una técnica que se conoce como *sandboxing* o *sandbox* –delimita una sección de código mediante la inserción de rutinas, que comprueban el espacio accedido durante su ejecución y tienen como resultado un control absoluto del flujo incluso frente a anomalías no previstas– [39] [40].

Algunos sistemas de información y por extensión los sistemas embebidos, ofrecen la capacidad de establecer dimensiones jerárquicos de ejecución –también denominadas como modos de ejecución–, tal que permite agrupar las capacidades que tienen los procesos –restricciones y privilegios–, así como su comportamiento sobre los distintos recursos de la plataforma, bajo contextos que facilitan la gestión. La mayoría de arquitecturas presentan al menos dos modos de ejecución, aunque existen excepciones que recogen más dimensiones, como puede ser el modo *hypervisor* –para la presentación de recursos virtuales–. El primero y

de mayor prioridad se denomina modo privilegiado, bajo el cual se ejecutan todos los procesos de confianza del sistema, tal que constituyen el *kernel*, y los cuales tiene acceso directo al *hardware* subyacente, por lo que se encargan de gobernar y administrar las capacidades y accesos a recursos. El segundo se refiere al modo usuario, donde un proceso ve restringido los elementos disponibles y accesibles, y en ocasiones necesitará valerse de las tareas ejecutadas en modo privilegiado para completar su función [41]. Así pues, las técnicas de protección de memoria, mayoritariamente son accesibles únicamente bajo el dominio de protección privilegiado –aunque bajo los denominados *microkernels* la gestión de memoria se rige desde el modo usuario, en oposición a los anteriores [42]–, de forma que todos los procesos ejecutados bajo el modo usuario, no podrán alterar la configuración del mecanismo de protección, puesto que la labor recae sobre una tarea privilegiada perteneciente al *kernel*, como único conocedor de las características de los recursos a nivel físico en la plataforma y su posición en el espacio de direcciones.

Además, resulta necesario distinguir aquellas implementaciones de protección de memoria que ofrecen garantías en tiempo estático –compilación–, de las que lo hacen durante la ejecución de la plataforma –denominadas como dinámicas– y del mismo modo, se establece una división entre las técnicas que soportan la protección de memoria en base a criterios *software*, *hardware* o una combinación de las anteriores. De entre los mecanismos de protección, las implementaciones *software* presentan una mayor flexibilidad, ya que permiten su modificación o mejora incluso ya desplegadas, pero llevan asociados problemas de sobrecarga en los tiempos de ejecución y se demuestran poco efectivos frente a fallos *hardware* inducidos por el entorno. Cabe señalar que la concepción de mecanismos de protección *software* basados en la incorporación de comprobaciones estáticas [43] [44], suelen presentar una ventaja frente a aquellas que por ejemplo, realizan la inserción de subrutinas en las llamadas a procedimientos (*sandboxing*). Por otra parte, las técnicas *hardware* a pesar de presentar un mayor consumo energético en la plataforma y costes recurrentes en la producción, se reconocen como alternativas efectivas frente a fallos generados intrínsecamente y extrínsecamente. Sin embargo, no permiten su modificación o ampliación bajo demanda, aunque si su configuración, ofreciendo así un comportamiento pseudodinámico –sufren de un incremento en los tiempos de ejecución para cada configuración–. Estos últimos presentan el mayor interés para la elaboración de este trabajo, dado que su complejidad es reducida frente a mecanismos *software*, en cuanto que no requieren de complejas heurísticas o de modificación en el comportamiento de herramientas de construcción *software*.

2.4 Mecanismos de protección de memoria existentes, características, discusión y elección

En el siguiente apartado se tratarán algunos de los mecanismos de protección *hardware* presentes en la actualidad.

2.4.1 Segmentación

La segmentación es un mecanismo de protección, que permite el uso del espacio de direcciones virtuales –este término entendido en el sentido de apariencia o no real–, esto es, se trabaja con los objetos que representan la aplicación sin necesidad de conocer su posición real en el espacio de direcciones físico. De esta manera, dichos objetos se encuentran contenidos en regiones con identificador único –direcciones contiguas del espacio de direcciones virtual para las que se define un tamaño variable– a las que se denominan segmentos. Puesto que no se manipulan direcciones reales, sino aparentes, el acceso se delega en última instancia a un módulo *hardware* específico –la protección reside en este mecanismo y no en el proceso de traducción de direcciones *per se*– encargado de producir una equivalencia entre la dirección accedida en el espacio virtual y su correspondiente dirección físifca, que recibe el nombre de unidad de gestión de memoria

o *MMU* (*Memory Management Unit*) –aunque históricamente la segmentación no siempre ha requerido de esta unidad, véase [45] [46]–.

La *MMU* recibe como entrada una dirección que codifica una estructura conocida como tabla de segmentos y el desplazamiento sobre esta, tal que permite la selección de la entrada correspondiente, las cuales contienen la dirección física de inicio de los segmentos de tareas. Una vez determinado el segmento objetivo, se sustituye el fragmento de la dirección lógica, correspondiente al inicio de la tabla de segmentos empleada, por la dirección física que da inicio a dicho segmento, de forma que el desplazamiento original no sufre modificación, dado que se encarga de indicar la posición del objeto dentro de este. Durante el proceso de traducción de direcciones, se comprueba que el desplazamiento realizado sobre el segmento pertenezca al rango límite establecido para este y además se validan los permisos definidos sobre este espacio para la operación efectuada. Es posible combinar la segmentación con otros mecanismos de protección tales como la paginación de memoria –como combinación de dos mecanismos *hardware*– y *sandboxing*, o por el contrario su extensión mediante la aplicación de dominios de protección, donde cada dominio provee a los segmentos contenidos de diferentes permisos en operación [47]. La asignación de tamaños arbitrarios para esta técnica induce a un fenómeno conocido como fragmentación externa, por la cuál existe suficiente espacio disperso para albergar nuevos objetos, pero no es posible su disposición de forma contigua para su asignación bajo un nuevo segmento. La compactación permite mitigar este efecto pero en ocasiones supone un coste no permisible para la ejecución. [41]

2.4.2 Paginación

Al igual que para la segmentación, la paginación soporta el uso de direcciones virtuales, por lo que en la misma medida demanda un mecanismo de traducción para esta tarea. Sin embargo, la paginación difiere de la anterior, en cuanto que divide el espacio de direcciones virtuales en bloques de tamaño fijo, los cuales reciben el nombre de páginas –donde las direcciones que conforman una página se suponen contiguas– y cada una de ellas presenta una correspondencia directa con bloques denominado como marcos –bajo los cuales residen los objetos que conforman la aplicación–, tal que comparten la misma extensión pero se limitan al espacio de direcciones físicas.

Con objeto de acelerar la ejecución y dada la discrepancia en velocidad para la jerarquía de memorias, se plantea la existencia de registros de procesador –tal que muestran la menor latencia para la ejecución de toda jerarquía–, los cuales conforman las entradas de tabla de página dado un proceso. No obstante, este enfoque resulta inviable, debido como principal razón al elevado coste en la creación de tales registros para aquellos sistemas que requieren de un considerable número de páginas por tarea. Por ello, es frecuente encontrar un formato de estructuras de datos en representación de dichas tablas, las cuales por norma deberán residir en niveles superiores de la jerarquía –con menores costes económicos y tecnológicos asociados–, en concreto la memoria principal. Algunas implementaciones ofrecen un registro cuyo contenido es la dirección base de la tabla referida con anterioridad, logrando así retener cierta mejora, en lo que respecta al incremento de latencias. Además, es frecuente la existencia de una *caché* denominada como *Translation Look-aside Buffer* (*TLB*), donde los accesos capturados por esta participan en la misma medida en la protección de la memoria y cuyas entradas,que consisten en un par clave-valor, alojan el resultado en traducción de aquellas páginas recientemente accedidas, de forma que la ejecución evita los costes en tiempo asociados a los accesos, siempre y cuando se produzca una coincidencia, esto es, la página accedida se encuentra presente en una entrada de *caché*. [41]

La protección para la aproximación se define para cada entrada de la tabla de páginas, en concreto, una

serie de bits codifican las operaciones permitidas sobre un rango del espacio. En ocasiones, se incluye el denominado bit de validez, tal que indica si una página corresponde al contexto de ejecución de un proceso, esto es, al conjunto del espacio virtual asignado a este. La paginación consigue un rango de precisión en la protección superior al presente en otras propuestas, pero se ve lastrado por un fenómeno de fragmentación interna, en la que parte del espacio asignado a la página –por tanto del marco– carece de uso. Es habitual que los procesos no hagan uso del espacio completo otorgado y en consecuencia, no sería obligatoria la creación de una estructura que cope el máximo número de entradas posible –hecho que reduce su peso en memoria–, sino que por contra, dadas las necesidades de la tarea, se impone una restricción en su tamaño, el cual además se compara con la dirección virtual accedida, de modo que al exceder dicho límite, se aborta la ejecución. La granularidad establecida para el mecanismo se considera inmutable en tiempo de vida del sistema –su valor se provee en tiempo de compilación–, hecho de relevancia, dado que la elección de tamaño de página condiciona por un lado, la fragmentación del sistema o por el contrario, el rendimiento en ejecución, puesto que a menor tamaño de página, se reduce la fragmentación, pero se producen mayor número de reemplazos en registros y estructuras, y viceversa.

2.4.3 Unidad de protección de memoria

De la misma manera que bajo la segmentación, la unidad de protección de memoria establece regiones del espacio de direcciones físicas, cada una con su propio identificador, cuyo número total disponible para la tarea viene determinado por la implementación *hardware* del mecanismo. Del mismo modo, la variedad de arquitecturas definen características tales como la existencia de regiones específicas para memoria de instrucciones –alberga en exclusiva lógica de la aplicación–, memoria de datos –objetos *software*– o regiones mixtas –si el código y los datos de la aplicación residen bajo el mismo espacio–. Para cada región, se define tanto la dirección de inicio como la extensión para el rango de direcciones que cubre –en ocasiones, imposiciones de la arquitectura para el tamaño de región conllevan a la fragmentación interna para el espacio–, indicando aquellas operaciones permitidas o en su defecto restringidas bajo este.

A diferencia de otros mecanismos, la particularidad para la aproximación es el solapamiento de regiones, tal que las operaciones permitidas para un punto resultan de la combinación de permisos individuales, pertenecientes estos a las regiones que lo contienen, donde el orden de aplicación y prioridad de región viene determinado por la arquitectura. Asimismo, es posible la asignación, dada una determinada tarea para la ejecución, de una o varias regiones para la representación del espacio de protección, que constituyen el denominado dominio de protección, tal que se intercambia bajo demanda y cuya implementación tiene lugar, o bien mediante un mecanismo *hardware* soportado por la arquitectura, cuyo efecto supone la aceleración para el cambio de contexto, o bien como aproximación *software*, la cual incrementa las latencias para la ejecución, pero supone menor coste recurrente.

Por último, cabe destacar que algunos fabricantes permiten la presencia de múltiples implementaciones del mecanismo, donde cada una presenta una función y un espacio de aplicación diferente –por ejemplo, una comprueba los accesos producidos por la unidad de procesamiento, mientras que otra lo hace para un maestro del sistema–, por lo que en lo relativo a este trabajo, tan sólo se indicarán aquellas que evalúan las operaciones iniciadas por la unidad de procesamiento.

2.4.4 Sistema de protección *Mondrian*

Toda la información recogida se encuentra disponible en el artículo original [48]. Se trata del mecanismo más característico para los aquí presentados, el cual abarca un espectro de componentes *software* y *hardware* para su realización, que además bebe de particularidades presentes en otros mecanismos tales como,

coexistencia de dominios de protección, espacio de direcciones lineal, división del espacio en regiones de precisión arbitraria –las cuales definen las acciones soportadas–, traducción opcional de direcciones –posible extensión– o existencia de jerarquías de memoria y componentes *hardware* específicos para la aceleración en su tarea. Soporta asimismo la integración con otros mecanismos de protección y en consecuencia bajo las diferentes arquitecturas –modificaciones efectuadas para este efecto pueden exceder ratificaciones estandarizadas de las mismas– que los definen.

La estructura principal del sistema de protección *Mondrian* recoge dos registros de control, bajo jerarquías de memoria asociadas a la unidad de procesamiento, de forma que el primero indica el dominio de protección asociado a la tarea en ejecución, mientras que el segundo contiene la dirección que apunta a la denominada tabla de permisos, la cual resulta única para cada dominio y cuya representación se elabora mediante una estructura conocida como tabla de segmentos, tal que un segmento –su significado difiere respecto del mecanismo de segmentación– constituye el rango de direcciones para los que la tarea tiene capacidad de modificar sus permisos en operación. Se identifican bajo el término bloque, el cual guarda una estrecha relación con respecto a las región identificada para la *MPU*, a aquellos segmentos para los que el resto de la división entre su tamaño –considerado potencia de dos– y la dirección de inicio –alineada respecto a este valor– resulta cero –es posible la existencia de segmentos no alineados en memoria, mediante una variación en la representación de segmentos–. La tabla de permisos reside bajo la memoria de sistema, por lo que del mismo modo que para la segmentación y la paginación, se requiere la disminución del coste que supone en tamaño la codificación de las zonas de protección, así como la sobrecarga en tiempos de ejecución, asociada a la comprobación de cada uno de los permisos en acceso a una posición del espacio. Por ello, se establecen tres aproximaciones que abordan su concepción, tal que cada una consigue un balance entre la complejidad de codificación para los permisos y el espacio empleado.

Debido al deterioro en rendimiento, que resulta de accesos para diferentes niveles en la jerarquía de memoria, se proveen dos estructuras intermedias que permiten mitigar estos efectos, en concreto una *caché* que recibe el nombre de *Permission Lookaside Buffer* (*PLB*) y una colección de registros denominados *sidecar* –como extensión de los registros asociados a la unidad de procesamiento y definidos por la arquitectura–. Un registro *sidecar* proporciona la suficiente información para la resolución de permisos con independencia de otras estructuras, reduciendo así las consultas efectuadas, pero si la referencia no se encuentra disponible, se requiere el acceso al segundo nivel en la jerarquía, esto es el *PLB*, que recoge además para cada entrada el domino al que pertenece. En última instancia, si no se encuentra una entrada válida en el *PLB*, entonces se accede a la tabla de permisos y con objeto de optimizar sucesivas comprobaciones, los datos obtenidos se cargan tanto en el registro *sidecar* correspondiente como en el *PLB*.

Para concluir, diferentes dominios de protección permiten generar vistas sobre el espacio de direcciones, de modo que es posible la repetición de permisos entre diversas tareas, en cuyo caso es posible evitar la duplicación de entradas que codifican dicha información, por ejemplo, empleando la misma tabla de segmentos para todos los dominios. Además, presencia de diferentes dominios por tarea implica recuperar la tabla de permisos para cada uno, con objeto de constituir la vista que proporciona para la protección, lo que supone un coste añadido asociado a cada cambio de contexto –efectuado mediante llamadas de sistema y por lo tanto una solución *software*–, aunque cabe la posibilidad de acelerar dicho proceso mediante el uso de *hardware* exclusivo para la tarea.

2.4.5 Discusión y elección de mecanismo

Una vez presentadas sumariamente las cualidades que definen a los sistemas de protección *hardware* aquí descritos, se procede a la elección de aquel indicado, comparativa mediante, dadas las necesidades y particularidades que presentan los sistemas empotrados y de cuyo resultado parte el resto del trabajo elaborado. En concreto, las propiedades empleadas en la tarea de decisión comprenden: disponibilidad a nivel comercial, la flexibilidad del sistema de protección (condicionada por la precisión –como espacio que abarca una región y expresado en bytes como unidad de medida– y el número de ventanas de protección), sobrecarga en el tiempo de ejecución y *hardware* requerido, para las cuales se recoge además un breve apunte:

- Disponibilidad comercial, propiedad para la que cual comentar el estado actual de los mecanismos para la protección. Por un lado, la segmentación en la actualidad no se considera como una opción autónoma para sistemas embebidos de alto rendimiento y soluciones genéricas, de modo que se releva a una combinación –incentivada por motivos de compatibilidad– junto a la paginación, la cual a diferencia, sí se encuentra presente de manera independiente en arquitecturas como ARMv7-A [49]. En lo que refiere a *Mondrian*, no se identifica ninguna solución comercial que lo implemente, por lo que su estudio queda relegado a una solución personalizada mediante el diseño lógico, hecho que descarta esta opción para el trabajo, en cuanto que excede las competencias previstas –no obstante, resulta de interés continuar la comparativa de sus características, con el fin de probar su valor frente al resto de mecanismos–. En cuanto a la unidad de protección, dada la popularidad de los sistemas embebidos basados en microcontrolador –aquellos con mayor aplicación comercial–, los cuales presentan restricciones de consumo y recursos, este resulta el mecanismo más abundante y extendido, ya que gracias a sus capacidades, hacen viable la adaptación a los cambios y necesidades que presentan los proyectos para este ámbito.

- Flexibilidad, propiedad para la cual *Mondrian* presenta la mayor ventaja, dada la diferencia entre la mínima precisión soportada –siendo esta el tamaño de palabra de la implementación– y el máximo valor –para el que no se establece límite, aunque no obstante, deberá adecuarse a las capacidades que ofrece la arquitectura–. Sin embargo, el número máximo de regiones para la protección, al igual que para la paginación y la segmentación –ambas sin las ventajas en precisión de *Mondrian*–, está condicionado en gran medida por el tamaño de la estructura descriptora empleada, hecho a tener en cuenta, puesto que reside en la memoria principal con las connotaciones que ello implica –respecto al uso de un recurso escaso para aquellos sistemas basados en microcontrolador, así como las diferencias en latencias de acceso que supone–. Tanto la paginación como la segmentación presentan problemas respecto del valor para la extensión en la protección, de forma que la primera requiere de la selección correcta para el tamaño de página, el cual es fijo, mediante un estudio de impacto, mientras que la segunda presenta problemas intrínsecos de la asignación de valor arbitrario –fragmentación externa–. La *MPU* presenta la mayor desventaja en este aspecto, puesto que se definen límites para la precisión –aunque esta presenta gran variabilidad en función de la plataforma–, así como para el número de regiones, generalmente reducido, impuestos por la arquitectura.

- *Hardware* requerido, puesto que son necesarios mecanismos de soporte para todas las opciones propuestas. Para aquellas que permiten el espacio de direcciones lógicas, resulta indispensable la presencia de *hardware* encargado de la traducción, así como de un elemento capaz de determinar si un acceso consta de los permisos requeridos, ya que una implementación exclusivamente *software* es inviable dada su complejidad. *Mondrian*, al trabajar con direcciones lineales, esto es, la dirección accedida es la misma que aquella para la que se comprueban sus permisos, se trata de un caso particular,

ya que no demanda un mecanismo para la traducción, pero sí necesita de la decodificación de direcciones, puesto que estas determinan las entradas para la tabla descriptora. Del mismo modo, la unidad de protección, conocedora del espacio de direcciones físicas, no se sirve de mecanismos adicionales para la traducción, sino que únicamente precisa de lógica que verifica la corrección de las operaciones. Todas las propuestas, de una forma u otra, se integran con el módulo de excepciones y en ocasiones, lo hacen con aquel dedicado al control de estado de la plataforma. La *MPU* a diferencia del resto de aproximaciones –las cuales requieren de estructuras para la gestión y representación de permisos, así como de técnicas que mejoran la velocidad en los accesos ante las discrepancias para la jerarquía de memoria–, tan sólo requiere de un mecanismo para el control y la configuración, cuya realización se refleja como un conjunto de registros de procesador o como un dispositivo mapeado en el espacio de direcciones.

- Sobrecarga en tiempo de ejecución, la cual se encuentra estrechamente ligada a la propiedad anterior. El impacto de esta característica se mitiga por un lado, en existencia de *hardware* específico para la comprobación de permisos, traducción o decodificación, según se requiera, y por otro, mediante técnicas de aceleración presentes en los diferentes niveles de la jerarquía de memoria –significativo para sistemas embebidos de alto rendimiento y propósito general–. En concreto, aquella basada en *MPU* resulta la mejor parada para este aspecto, en cuanto que su implementación se consigue a través de lógica sencilla sin presencia en varias jerarquías y puesto que sólo necesita la codificación para su manejo, así como una rutina para el tratamiento en caso de condición anormal en el flujo de ejecución, siendo estos los únicos elementos que contribuyen al incremento de latencias. En lo que respecta al resto de sistemas, la ausencia de entradas válidas bajo aquellas jerarquías próximas al procesador –*caché*, *TLB* o registros–, suponen el mayor incremento en tiempos de ejecución, siendo en alguno casos prohibitivos –en acceso a memoria persistente–. Además la presencia de estas técnicas deriva en una ejecución no determinista, que resulta inadmisible bajo el ámbito de los sistemas embebidos, en especial para aquellos con requerimientos de tiempo real. Al igual que para la *MPU*, estos demandan de rutina para la configuración y tratamiento en caso de excepción.

La existencia del espacio de direcciones lógicas resulta de utilidad para sistemas complejos, por ejemplo, los denominados sistemas de propósito general o los sistemas embebidos de alto rendimiento –aunque también viable para plataformas con limitados recursos en cuanto a memoria–, donde existen un elevado número de tareas que conforman la aplicación, de forma que cada una sin perturbar al resto remite a su espacio asignado. Sin embargo, las imposiciones sobre latencias en cambio de contexto y el incremento en consumo energético dada la presencia de lógica *hardware* adicional para la tarea, suponen un impedimento en su adopción para ámbitos que requieren bajos tiempos de respuesta y para los que tan sólo se requiere un reducido número de tareas –conocido *a priori*–, los cuales se benefician de la manipulación directa del espacio de direcciones físicas, así como de aproximaciones con carácter estático. Generalmente, las plataformas de sistemas empotrados basados en microcontrolador ofrecen servicio para este marco de aplicación y puesto que estas no incorporan una *MMU* como requisito indispensable para la traducción, se descartan como opción los denominados mecanismos con soporte de direcciones virtuales –paginación y segmentación–. El sistema *Mondrian* resulta la opción más atractiva de entre todas las propuestas, dadas las ventajas que ofrece al tratarse de un mecanismo personalizable –que incluye configuración del nivel de lógica requerido para su representación o integración junto con mecanismos previamente existentes y complementarios–, pero a pesar de ello, se descarta dada la complejidad que presenta su concepción como una solución a medida, hecho que desborda por completo los objetivos propuestos. Se concluye de esta manera, que la protección basada en unidad de protección es la única opción que se ajusta a los intereses del trabajo y por ello, la opción escogida para la elaboración de la propuesta, de modo que en lo sucesivo, se abordan algunas de las implementaciones comerciales y sus capacidades.

2.5 IMPLEMENTACIONES CON APLICACIÓN COMERCIAL ACTUALMENTE EXISTENTES

Una vez establecida la unidad de protección de memoria como el mecanismo más indicado para las protección de sistemas embebidos basado en microcontroladores, se procede a la exposición de varias implementaciones cada una definida en una arquitectura que además se realizan a través de procesador comercial. De entre todas las posibilidades se han seleccionado aquellas de vigencia actual (se siguen manufacturando y los fabricantes proporcionan soporte en forma de documentación y herramientas de desarrollo actualizadas en el momento en el que se realiza este trabajo). Además se ha considerado incluir una implementación definida bajo la arquitectura *RISC-V* cuyo módulo de protección se concibe como un mecanismo que evalúa operaciones sobre el espacio de direcciones físicas, pero que a diferencia del resto permite su integración con un módulo de traducción para el espacio de direcciones lógicas. A pesar de esta particularidad, se han comprobado una semejanza con la funcionalidad de una *MPU*, que sumado al interés que presenta esta arquitectura en los últimos años la convierten en una alternativa en el sector tecnológico muy atractiva.

Para cada unidad de protección se recoge la terminología definida por la arquitectura y/o fabricante que en lo sucesivo se refiere a ella como terminología propietaria (incluida en la documentación disponible para cada una de ellas, por ejemplo el nombre que identifica a un registro) para posteriormente unificar los elementos comunes bajo un formato que se referirá como operaciones básicas que representan las capacidades abstractas de las unidades de protección de memoria (es decir aquellas operaciones que no difieren entre sí y presentes en todas las implementaciones identificadas para las aquí tratadas).

Elegida la opción conformada por *MPU* como el mecanismo más adecuado en la tarea de protección, se procede a la exposición de varias implementaciones cada una definida para una arquitectura. Entre el producto comercializado y la arquitectura en la que se inspira, se encuentra una especificación intermedia a la que se denomina como microarquitectura, que generalmente recoge detalles específicos de la implementación para aspectos tales como número máximo de regiones o mapa por defecto de memoria entre otros, pero bajo la cual no se definen con precisión características cuya decisión recae en el fabricante tales como tecnología para la memoria del dispositivo o periféricos disponibles. El uso de microarquitecturas como propiedad intelectual ofrece diseños ya preparados, de manera que estos tan sólo necesitan integrar aquellas extensiones requeridas para la creación de un microcontrolador comercial, reduciendo así los tiempos de fabricación. En concreto bajo este apartado, las posibles aproximaciones tratadas se recogen tanto por una arquitectura, lo que implica una equivalencia en el diseño de la unidad de protección para todas y cada una de las implementaciones que refieren a esta, así como por una microarquitectura, en cuyo caso sólo compartida por aquellos productos que derivan de esta.

De entre todas las posibilidades se han seleccionado aquellas de vigencia actual, es decir, se siguen manufacturando y los fabricantes proporcionan soporte en forma de documentación y herramientas de desarrollo actualizadas, para el momento en el que tiene lugar el trabajo. Además, se ha considerado oportuno incluir la aproximación definida bajo la arquitectura *RISC-V* [50] ya que a diferencia del resto, su módulo de protección permite la integración con un mecanismo para la traducción de direcciones lógicas. A pesar de esta particularidad, su funcionamiento es semejante al del resto de unidades de protección, hecho que, sumado al interés que presenta esta arquitectura en los últimos años –no obstante, sufre de una lenta adopción–, la convierten en una alternativa en el sector tecnológico muy atractiva.

Cada unidad de protección presenta terminología particular disponible en su documentación, que en lo sucesivo se referirá como terminología propietaria –por ejemplo, el nombre que identifica a un registro con-

creto–, con el fin de unificar posteriormente aquellos elementos comunes, bajo el formato de operaciones primitivas, que representan las capacidades abstractas de las unidades de protección de memoria, es decir, aquellas características que no difieren entre sí y presentes en todas las aquí expuestas.
Se precisan a continuación los módulos junto con la exposición de sus particularidades, así como su funcionamiento:

2.5.1 ARQUITECTURA *RISC-V - Physcal Memory Protection (PMP)*

Con objeto de simplificar la labor, la descripción en este caso se limita a la versión de arquitectura para el ancho de palabra de 32 *bits*. La arquitectura dispone de tres modos para la ejecución, que afectan tanto a la manipulación de la unidad de protección como a la comprobación de permisos en acceso a direcciones del espacio físico, a saber: máquina (*M*), supervisor (*S*) y usuario (*U*), y ofrece un sistema de interrupción, que se conjuga con el mecanismo a describir, tal que permite identificar el origen y la causa del error en acceso.
En lo sucesivo, se emplearán registros para la descripción y configuración del módulo, cuya representación además se supone propia de la implementación –por ejemplo, mapeados en el espacio de direcciones o bajo un *coprocesador*–, pero reciben una especificación genérica a la que adherirse. El primer tipo se expresa como una serie de entradas identificadas como *pmp**n**cfg* y de tamaño ocho *bits*, donde la ***n*** representa el número total de regiones –siendo 16 el máximo soportado–, las cuales son agrupadas bajo un segundo conjunto de registros de gran densidad, denominados como *pmpcfg**x***, tal que la ***x*** representa un ordinal respecto del número total de estos –variable según la longitud de palabra considerada por defecto– y accesibles únicamente desde el modo máquina. Dichos registros *pmpcfg**x*** se comportan además como dominios, en cuanto que compendian información asociada a las regiones, facilitando así su edición en cambio de contexto.

Cada entrada para la configuración presenta la siguiente forma:

- *Address Matching* (*A*), tal que permite seleccionar el tipo de codificación en coincidencia de dirección. Dada pues una posición de acceso, se determina qué parte de la información codifica el rango de la ventana de protección y cuál hace referencia a la dirección de inicio, logrando así un alto grado de flexibilidad para la conformación de regiones de protección. Las opciones válidas son: *OFF*, *TOR*, *NA4* y *NAPOT –NA4* como caso particular de *NAPOT*–.

- *Locking* (*L*), indica que la entrada se encuentra bloqueada, es decir, una vez activo no se permite la modificación de contenido bajo el modo máquina, requiriendo para ello el reinicio de la plataforma. De esta manera, los permisos que afectan a los modos supervisor y usuario, también aplican al modo máquina.os supervisor y usuario, también aplican al modo máquina.

- Permiso de ejecución (*X*), de modo que habilita la extracción de contenido de una posición, con motivo de su posterior ejecución por la unidad de procesamiento. En caso de fallo –permiso revocado– se genera la correspondiente excepción.

- Permiso de escritura (*W*), tal que admite la modificación del valor para una dirección de memoria. En caso de fallo –permiso revocado– se genera la correspondiente excepción.

- Permiso de lectura (*R*), el cual contempla la consulta para el contenido de una posición. En caso de fallo –permiso revocado– se genera la correspondiente excepción.

A cada entrada –por tanto región de protección– se le asocia un registro *pmpaddr**n*** de 32 *bits* –donde ***n*** equivale al correspondiente número de región– capaz de codificar la dirección límite para la ventana seleccionada, su extensión o ambas a la vez. El registro representa los *bits* [33:2] de la dirección –el espacio de

protección efectivo equivale a 2^{34} *bits*–, puesto que los contenidos se representan siempre en posiciones alineadas con múltiplos de cuatro *bytes* –siendo esta la mínima precisión admitida y semejante a la capacidad ofrecida por *Mondrian*–, de manera que los dos *bits* menos significativos quedan sin representación y por ende con valor nulo. El comportamiento asociado a su codificación, es dependiente del campo *A* escogido para la entrada de configuración, toma la forma descrita en las siguientes posibilidades:

- En opción *OFF*, la región de protección no tiene efecto y por tanto deshabilitada.
- Para la opción *TOR*, el registro de dirección codifica el límite superior para el rango, mientras que el referido a la entrada precedente define el límite inferior –para la entrada 0 se supone el comienzo del espacio de direcciones–. La extensión del rango se obtiene como la diferencia entre los limites superior e inferior, sin restricción en su alineación, por lo que se permite un tamaño arbitrario.
- Respecto a *NAPOT* y su caso particular *NA4*, la codificación refleja tanto el inicio de la ventana como su tamaño, alineado este con direcciones potencias de dos, aunque para *NA4* no se requiere especificar dicho valor, puesto que siempre viene dado implícitamente como el mínimo soportado (cuatro *bytes*). Para *NAPOT*, la extensión del rango se expresa como una serie de unos sucedidos por un cero, cuya posición indica la precisión en *bytes* –mediante la fórmula 2^{G+2}, siendo *G* tal posición– y que además separa la parte que constituye la dirección de inicio para la región.

Tal y como se observa, no se distinguen entre regiones exclusivas de código y de datos para la asignación de permisos, los cuales se comprueban bajo cualquier circunstancia, incluso para accesos que encuentran su ejecución determinada, esto es, aquellos cuya acción tiene lugar en evaluación de condición. Si un acceso tiene origen en modo máquina y no existe ninguna entrada con restricción para este, entonces se completa por defecto con éxito. Asimismo, si este se realiza bajo los modos supervisor o usuario y hay al menos una entrada configurada, en ausencia de coincidencia –la dirección no se encuentra contenida en la región– se arrojará un fallo. Se permite el solapamiento de regiones, tal que la prioridad para su aplicación viene dada por orden, es decir, aquella con la numeración más baja, de entre todas las regiones que reaccionan, proporciona los permisos.

Por último, cabe señalar que la tarea de asignación de atributos referidos al comportamiento de la memoria –por ejemplo, si un acceso se interpreta como atómico– no recae en la unidad de protección, sino que se determina por separado, pero en lo que respecta a su comprobación, ambas suceden de forma paralela. En este caso, aquellos accesos que debido a los atributos de memoria no puedan completarse en una única operación, como es el caso para accesos no alineados, se dividen en tantas como sean necesarias, donde para cada una se aplica la evaluación de permisos, por lo que es posible la aparición de un fenómeno, de forma que algunas operaciones para un mismo acceso se completan con éxito, mientras otras son revocadas.

2.5.2 ARQUITECTURA *PowerPC* MICROARQUITECTURA *NXP e200z4* [1]

La unidad de protección aquí descrita se define por una microarquitectura de fabricante, la cual se construye a partir de la arquitectura *PowerPC*, de manera que cualquier plataforma derivada de ella comparte todas las características aquí recogidas y donde aquellas instrucciones, que habilitan la interacción con el módulo de protección, son parte exclusivas de esta. No obstante, la implementación incorpora elementos recogidos por la arquitectura, tales como interrupciones –*Data Storage Interrupt* e *Instruction Storage Interrupt*– para la notificación en acceso ilegal, así como el uso los modos de ejecución siendo estos: supervisor y usuario. El término región, representado como una entrada descriptora –de un total de 24, que a su vez determinan el número máximo permitido para dichas regiones– contenida en una tabla, se concibe como subconjunto del

espacio de direcciones de tamaño arbitrario, para el que se establece la comprobación de permisos. El acceso para su configuración es indirecto, esto es, mediado por cuatro registros *MAS**n*** (*MPU Assist Registers*) –donde la ***n*** indica el registro concreto al que se refiere comenzando por *MAS**0***–, que asisten en la tarea, puesto que las entradas no se encuentran mapeadas en el espacio de direcciones de la plataforma. Además, se definen instrucciones específicas con este fin tales como *mtspr/mfspr*, que habilitan la escritura y lectura de los registros *MAS**n***, y *mpuwe/mpure*, que permiten escribir el valor en *MAS**n*** como nueva configuración para la *MPU*, así como leer la configuración existente –volcando el contenido en *MAS**n***–, respectivamente.

La distribución de entradas para la tabla se advierte de la siguiente forma:

- Seis regiones de protección dedicadas al espacio que contiene el código de la aplicación.
- 12 regiones destinadas en exclusiva al espacio de direcciones de datos.
- Seis regiones adicionales sin aplicación definida a priori, que puede configurarse bajo demanda y excluyente respecto de un espacio una vez se define su función.

Cada entrada se conforma por una serie campos, tal que se recopilan a continuación:

- *VALID*, el cual determina si la región participa en la comprobación de permisos.
- *UPPER_BOUND*, como límite superior comparado con la dirección para el acceso.
- *LOWER_BOUND*, como límite inferior comparado con la dirección para el acceso.
- *TID[0:7]*, tal que asocia una tarea y una región de protección concreta (segmentación).
- *TIDMSK[0:7]*, como máscara que permite la asociación de una región a múltiples tareas.
- *UMASK*, como máscara para los cinco bits más significativos de la dirección accedida.
- *INST*, que determina el tipo de región –dato o código– para las entradas personalizables.
- *DEBUG*, de modo que habilita la generación de eventos en depuración
- *SXSWSR*, como permisos de ejecución, escritura y lectura respectivamente para el modo de privilegio supervisor.
- *UXUWUR*, como permisos de ejecución, escritura y lectura respectivamente para el modo de ejecución usuario.
- *I*, el cual inhibe el alojamiento en cache para posiciones contenidas en la región.
- *IOVR*, que sobrescribe el comportamiento anterior en coincidencia de región múltiple.
- *G*, tal que determinar si el acceso a datos se realiza de forma especulativa.
- *GOVR*, que sobrescribe el comportamiento anterior en coincidencia de región múltiple.
- *IPROT*, como protección frente a invalidación de entrada.
- *RO*, tal que deshabilitar cualquier configuración para la entrada –sólo lectura–.

En ausencia de región válida configurada, la cual provoca coincidencia en acceso, la operación no resulta efectiva, ya que es revocada por la *MPU* a la par que genera la correspondiente excepción, la cual deberá ser debidamente tratada. La comprobación de permisos solamente aplica para el primer *byte* accedido, por lo que si realiza un acceso a una dirección dispersa entre varias regiones, por ejemplo, aquella no alineada en memoria o para la que su extensión comprende una palabra doble, la operación se divide en múltiples acciones, tal que aquellas que inciden sobre regiones menos restrictivas llegan a completarse. Al igual que para *RISC-V*, esta aproximación permite la agrupación de regiones, aunque a diferencia de esta, se asocian respecto de una tarea a través del campo *TID*, estableciendo así un dominio de protección intercambiable en función de las necesidades del contexto de ejecución –semejante al mecanismo *Mondrian*–, de forma que se deniega el acceso si la tarea que realiza la operación no es la esperada por la vista actual del espacio. Además, con objeto de facilitar la adopción del dominio correcto en cambio de tarea, se permite designar a las regiones compartidas como inmutables –campo *RO*–, así como invalidar aquellas no requeridas mediante un mecanismo *hardware*, indicando previamente las solicitadas –modificando *IPROT* según sea necesario–.

Para concluir, en este caso la unidad sí define, a través de dos atributos, el comportamiento del espacio de direcciones, tal que el primero (*inhibit*) refiere tanto a accesos destinados a datos como a código y determina si el contenido para las posiciones accedidas se aloja en línea de *caché*, tal que es accedido en lo sucesivo a través esta –con efecto sobre el determinismo en tiempos de ejecución–, mientras que el segundo (*guarded*) únicamente aplica a accesos sobre el espacio de datos y asimismo concreta si estos se completan de forma especulativa o por el contrario, en el orden establecido por el flujo de la aplicación. En solapamiento de regiones esta aclaración difiere, de forma que la entrada con permisos más restrictivos también proporciona el valor de guarded y por contra, aquella más laxa decide la inhibición de caché, siempre y cuando no se disponga al menos una entrada con los campos *GOVR* e *IOVR* precisados.

2.5.3 ARQUITECTURA *Infineon TriCore™* [2]

De nuevo, esta arquitectura incluye diversos modos de ejecución –identificados en terminología propietaria como niveles de privilegio–, siendo estos: supervisor, usuario-0 y usuario-1 –difiere del usuario-0, en cuanto que no puede realizar operaciones específicas en periféricos ni tampoco desactivar las interrupciones–, que tienen implicación en la protección de memoria. Con motivo de su configuración se emplean registros de control y estado, cuya posición se mapea en el espacio de direcciones del sistema. Esta arquitectura diverge del resto, dado que se trata de una arquitectura exclusivamente *Harvard*, hecho que determina el comportamiento de los accesos a datos y código, los cuales residen en espacios independientes, así como delimita la estructura del mecanismo de protección. Los permisos concebidos dependen del espacio de aplicación, de forma que aquel dedicado a albergar objetos *software* consiente la lectura y escritura, mientras que el destinado al código, sólo precisa la ejecución. La incorporación del concepto de dominio en su realización, la convierte en una de las propuestas más atractivas. El sistema de protección se define, bajo terminología

propietaria, como protección de memoria basada en rango, el cual se interpreta como un subconjunto del espacio de direcciones para el que se establecen permisos de acceso. Se define una colección de rangos para el espacio de datos y otra dedicada a la protección del código, donde para ambas se establece un límite de rangos implementados, que comprende entre cuatro y 16. Cada rango admite una precisión codificada como la diferencia entre dos registros (*Data/Code Protection Range Register Upper/Lower Bound*), los cuales representan el límite superior e inferior respectivamente–, cuyo valor mínimo soportado es de ocho *bytes* –los tres *bits* menos significativos para la dirección son irrelevantes–, mientras que el máximo se establece por el ancho de palabra de la arquitectura (32 *bits*). Esta propuesta introduce el concepto de dominio como conjuntos de protección –limitados a un mínimo de dos y un máximo de cuatro–, tal que agrupan una selección de

rangos –que admiten su compartición entre dominios– asociados a una serie de permisos y dispuestos en parejas del mismo tipo. Cada conjunto, que describe una vista instantánea del espacio de direcciones sujeta a un contexto de ejecución, presenta sus propios registros (*Data/Code Protection Set Configuration Register*), de modo que estos le permiten seleccionar los rangos del par a utilizar, así como la distribución de permisos. Su presencia en la arquitectura consigue una considerable deducción en la sobrecarga por cambio de contexto, puesto que la modificación de permisos tiene lugar a través de la recuperación del correspondiente conjunto, en contraposición a una actuación individual sobre cada rango. Una vez activo el mecanismo de protección y con al menos una región configurada para la opción de espacio indicada, si existe al menos una ventana, la cual comprende la dirección accedida y cuyos permisos otorgados se corresponden con el tipo de operación realizada, el acceso se completa con éxito. Ante la negativa, la plataforma produce una trampa, en términos propietarios, que ha de ser capturada y tratada antes de continuar con la ejecución, puesto que se trata de una situación anormal, que rompe con el flujo esperado. La aplicación de restricciones resulta independiente del nivel de privilegio que presenta el sistema en un instante concreto, por lo que si así se requiere, es posible la denegación de acceso, efectuado este desde el modo supervisor. Al igual que en

RISC-V, el comportamiento de los accesos, esto es, los atributos de memoria, se define con independencia del mecanismo de protección, pero repercute en este, en tanto en cuanto los atributos determinan si aplica o no la protección para los espacios que designan. Por ejemplo, regiones precisadas como periféricos o como áreas exclusivas que facilitan el intercambio de contexto, no participan en la comprobación de permisos por el mecanismo. Para accesos sobre regiones que se intersecan, la protección se consigue mediante disyunción lógica, de modo que si al menos uno de los rangos habilita la operación, esta se completa con éxito. Además, si un acceso abarca dos rangos distintos para los que no existe solapamiento, el resultado es indefinido.

2.5.4 ARQUITECTURA *Renesas RXv3* [3]

En cuanto a esta propuesta, se definen dos modos de ejecución: supervisor y usuario, cuya distinción es relevante, puesto que la protección aplica en exclusiva al modo usuario, hecho que constriñe su flexibilidad de uso respecto del resto. Dispone de una colección de registros de control y estado, sólo accesibles mediante una operación originada por el procesador –o la unidad *DMA*– y en modo supervisor, que se mapean en el espacio de direcciones físicas.

Bajo esta opción, se precisan nueve regiones, tal que una actúa como región de fondo, es decir, cubre el espacio completo de direcciones –sin posibilidad de editar su extensión– y que permite determinar el permiso de un acceso para el que no existe ventana válida, mientras que el resto constituyen el número máximo de regiones personalizables. Para cada una, se establece una precisión mínima, la cual recibe el término propietario de página (16 *bytes*), de modo que los cuatro *bits* menos significativos para la codificación de la dirección accedida son irrelevantes –se leen como cero–, mientras que la máxima extensión, la cual cubre el espacio completo de direcciones, viene dada por la amplitud de la palabra, esto es, 32 *bits*. Se destinan dos tipos de registros de control, siendo estos *RSPAGE**n*** y *REPAGE**n*** –donde ***n*** indica el número de región accedida–, los cuales definen respectivamente las direcciones –ambas alineadas con posiciones múltiplo de páginas, de modo que el resultado de su diferencia es también múltiplo de página– de inicio y fin de la ventana en el espacio de direcciones. Además, *REPAGE**n*** incluye campos auxiliares, que precisan la validez de la región, así como los permisos disponibles –ejecución, lectura y escritura– para esta. En caso de solapamiento, la información para el control de acceso se obtiene como disyunción lógica de las regiones involucradas –región de fondo incluida–, de modo que tiene preferencia la concesión de permisos frente a la revocación. El comportamiento para accesos no alineados, que cubren múltiples regiones de protección, se supone indefinido.

El control de permisos da comienzo tras la primera transición a modo usuario, por lo que la configuración de la unidad se produce inevitablemente en el arranque de la plataforma, ya que este sólo surge efecto cuando la opción global de protección –registro *MPEN*– se encuentra previamente habilitada y al menos una región activa válida se encuentra presenta. Una vez en funcionamiento, por cada acceso se consulta la información disponible, que afecta al resultado de la acción. Dicha información se obtiene de aquellas regiones que generan una coincidencia, es decir, encierran a la dirección accedida, así como la que proporciona la región de fondo. En extracción de instrucción, entonces se comparan los permisos asociados a la posición accedida con el tipo de operación, siendo este caso ejecución, mientras que en acceso a dato, se averigua si se trata de una lectura o escritura. La operación no se completa para cualquier punto que encuentra sus permisos revocados, de modo que se lanza una excepción –se designa en terminología propietaria como violación de protección–, que requiere tratamiento. Para facilitar la gestión de la excepción y reducir el tiempo tomado hasta retomar el flujo principal, se incorporan una serie de registros de estado, que detallan el tipo de acceso causante de la violación, la región que contiene la posición accedida, así como la dirección exacta que origina el fallo –en extracción de instrucción se emplea la pila y no registro–.

2.5.5 ARQUITECTURA *ARM®v7-R* MICRO-ARQUITECTURA [4] *Cortex™-R5F* [5]

Al igual que para el resto de implementaciones tratadas, la arquitectura define dos modos de ejecución a saber: supervisor y usuario –o en términos propietarios *PL1* y *PL0* respectivamente–, tal que demuestran una conducta específica tanto para la unidad de protección como en lo relativo a accesos, en función de la microarquitectura elegida. Se define un mapa de memoria por defecto, que designa *a priori* para una selección rangos con sus respectivos atributos y permisos, de forma que define un comportamiento predeterminado para el espacio de direcciones, en ausencia de una unidad de protección válida –implementada o por contra configurada y habilitada– y que resulta de utilidad para los fabricantes, puesto que delimita el espacio válido para el despliegue de módulos lógicos propietarios. Los registros para control, configuración y representación de estado son provistos bajo un módulo *hardware* denominado *coprocessor*, el cual asiste a la unidad central de procesamiento y cuya interacción tiene lugar mediante instrucciones específicas, que permiten indicar los correspondientes registros. A continuación se recogen los registros que participan en la descripción del mecanismo:

- *System Control Register* (*CTRL*), el cual recoge aquellos campos que afectan a la funcionalidad general del sistema, de los cuales, dos refieren a la unidad de protección, tal que habilitan la *MPU* y la región de fondo para esta respectivamente –en caso contrario aplicaría el mapa de memoria por defecto, sin restricción alguna en acceso–.
- *Data Fault Status Register* (*DFSR*), que posibilita la identificación de causa concreta en excepción de tipo *data abort* y de relevancia para el mecanismo aquella bandera, que indica la falta de permisos en operación.
- *Instruction Fault Status Register* (*IFSR*), tal que concreta el tipo de fallo en ejecución de instrucción, donde de nuevo resulta relevante el estado, el cual indica falta de permisos.
- *Data Fault Address Register* (*DFAR*), registro el cual captura la dirección cuyo acceso produce la excepción en el flujo y de interés para la rutina que maneja el fallo.
- *Instruction Fault Address Register* (*IFAR*), el cual detalla la dirección que identifica la instrucción cuya ejecución genera el fallo y de valor para la rutina que maneja el fallo.

- *MPU Type Register*, registro que alberga información acerca de la implementación de la unidad de protección. En concreto, detalla el número máximo de regiones, así como el modelo de memoria empleado –*Harvard* o *Von Neumann*–.
- *Memory Region Number Register*, que determina el rango de protección concreto para el cual hacen referencia el resto de los registros aquí descritos, en cuanto que el resto de ventanas y sus registros asociados, no son accesibles de forma simultanea, por lo que permanecen en un banco, a la espera de su configuración.
- *Region Base Address Registers*, registro que precisa la dirección de inicio para aquella ventana de protección seleccionada.
- *Region Size and Enable Registers*, el cual representa la flexibilidad de la ventana de protección elegida, ya que permite incluir o excluir subdominios de esta, el tamaño del rango que cubre, así como su validez.
- *Region Access Control Registers*, tal que determina el tipo de memoria y los permisos para las operaciones de acceso correspondientes a la región programada.

Cabe mencionar la existencia de registros, los cuales ofrecen soporte al sistema encargado de la gestión de recursos en la tarea de identificación de contexto (*CONTEXTIDR*), proceso (*TPIDRPRW*) e hilos de ejecución (*TPIDRURO*, *TPIDRURW*), de modo que es factible la creación de dominios de protección, representados como estructuras, que asocian una tarea con su correspondiente vista del espacio de direcciones y se restauran en cambio de contexto.

La arquitectura no exige a los fabricantes la presencia del mecanismo en su implementación, pero sí limita el número máximo de regiones de protección disponibles a 16, donde para cada una se especifica la precisión del rango, con 32 *bytes* de valor mínimo y máximo de 4 *GiB* –total del espacio–, así como la dirección base, a la que se impone una restricción, tal que su inicio parte de una posición múltiplo del tamaño asignado al rango de la ventana, por lo que el resultado de la operación de módulo dicha dirección y su extensión será cero. Se definen dos conjuntos de permisos –uno por modo de ejecución– para cada región, como una combinación de autorizaciones básicas representadas como campos de registros, siendo estas la capacidad de lectura y escritura. Tal y como se observa, la autorización para la ejecución no se incorpora a las anteriores, puesto que no depende de un valor de campo, sino de tres condiciones particulares, en concreto, el permiso de lectura no se encuentra revocado, la región no se define explícitamente como no ejecutable –*execute never* en términos propietarios– y el tipo de memoria es normal. Cada región asimismo se divide en un total de ocho subregiones con 256 *bytes* de tamaño mínimo soportado, las cuales mejoran la flexibilidad del módulo, en cuanto que su control resulta independiente entre sí, de modo que es posible excluir fragmentos del espacio con precisión, si así se requiere. Además de las constatadas propiedades, cada región se asocia a una serie de atributos, de forma que como se comprueba, las capacidades del módulo no se acotan a la verificación de corrección en acceso, sino que también define el comportamiento para el espacio de memoria. A continuación, se incorporan sumariamente los atributos contemplados por la arquitectura:

- Tipo de memoria (fuertemente ordenado, dispositivo, normal), atributo cuyas opciones son mutuamente exclusivas entre sí y cuya descripción se plantea en base a la capacidad de acción –o ausencia– concedida a la operación de acceso. El tipo normal se concibe para espacios, que esperan accesos recurrentes –implícitos o explícitos–, por lo que resultan útiles para el almacenamiento de información referida a datos o código de la aplicación. Accesos sobre este tipo resultan idempotentes si se requiere y asimismo pueden ser reordenados y agrupados, y del mismo modo, se permiten operaciones especulativas –único tipo que soporta este efecto–, así como el alojamiento en *caché*. Por contra, los tipos

restantes difieren respecto del anterior, en cuanto que no admiten más accesos que aquellos estipulados de manera explícita, para los que se espera un posible efecto secundario. Resultan de utilidad para representar secciones destinadas a periféricos, para los que el orden de acceso ha de ser inmutable y que suelen demandar mecanismos de sincronización. Cabe destacar que divergen en lo relativo a las operaciones de escritura, ya que en fuertemente ordenado las operaciones sólo se completan cuando alcanzan su destino, hecho no requerido para el tipo dispositivo.

- *Shareability/Non-shareability*, el cual comprende distintos significados para los tipos previamente identificados y que excluye al tipo fuertemente ordenado, puesto que *shareability* aplica a este por defecto. La opción identificada como memoria normal con *shareability*, describe qué espacios son accesibles simultáneamente para implementaciones con múltiples unidades de procesamiento, para las que se deberá garantizar la coherencia de la información contenida, esto es, todas observan los mismos datos para dicho espacio. En caso contrario, se determinan aquellos espacios accesibles de forma exclusiva por las unidades. Con respecto a memorias de tipo dispositivo, el atributo determina si la interfaz de acceso empleada por una unidad de procesamiento resulta o no compartida.

- *Non-cacheable*, que deniega el alojamiento en cualquier nivel de *caché* para el contenido correspondiente a la posición accedida y cuya aplicación sólo es soportada para memorias definidas como normal. Transversalmente, mantiene la coherencia de la información, puesto que la dirección que la contiene resulta única.

- *Read-allocate*, tal que indica si el dato accedido en operación de consulta puede alojarse en *caché*, si y solo si aún no se encuentra presente, de modo que posteriores accesos se registran sobre esta y no en el espacio de direcciones.

- *Write-allocate*, al igual que el anterior atributo determina si el contenido accedido es alojado en caso de fallo en *caché*, pero referido a la operación de modificación.

- *Write-throught cacheable*, el cual provee las características *read-allocate* a cualquier operación de consulta de contenido. En modificación, los cambios se propagan tanto a *caché* –si esta aloja al dato– como a la memoria principal del sistema. Se desestima la combinación con *write-allocate*, puesto que el volcado de información a *caché* es redundante, ya que siempre se accede a memoria con las consecuentes latencias.

- *Write-back cacheable*, del mismo modo que para el atributo que lo precede, se aplica *read-allocate* implícitamente, pero a diferencia de este presenta un menor número de accesos directos, dado que un dato sólo sufre modificación en memoria bajo dos condiciones. La primera supone la ausencia del bloque accedido en *caché*, en cuyo caso la modificación no tiene lugar para este nivel de la jerarquía, mientras que para la segunda sí se encuentra presente, pero el cambio aún no tiene efecto, de modo que si esta entrada es remplazada, entonces su contenido es volcado a la memoria.

Siempre que la protección de memoria se encuentre habilitada y exista al menos una región válida configurada, tendrá lugar la comprobación de permisos para cualquier acceso. En caso de existir varias ventanas con solapamiento, aquella considerada como prioritaria provee las restricciones, la cual se distingue bajo un proceso estático basado en orden numérico ascendente, es decir, la primera región registra la menor de las prioridades, mientras que la última la mayor. Si la dirección accedida no se registra bajo al menos una de las ventanas de protección o estas no presentan los requerimientos para el tipo de operación, entonces el sistema genera una condición anormal en el flujo –precisada por la arquitectura como fallo–, tal que se distinguen entre tres tipos para los que una rutina de servicio recopila la información necesaria, con objeto de restaurar la ejecución. El primero se refiere al fallo por región de fondo, tal que ocurre cuando se realiza

un acceso para el que no existe coincidencia de región alguna –si la región de fondo no se encuentra activa y la operación se inicia desde el modo privilegiado, entonces no se genera fallo–. Mientras, el segundo de ellos corresponde al fallo de permisos, que ocurre cuando una operación de acceso no cumple las restricciones que establece la región de protección sobre la posición accedida. Por último, el fallo en alineamiento, siendo este el más particular, el cual tiene lugar cuando la dirección de la posición accedida no se corresponde con el tamaño y alineamiento requerido –especificado por la arquitectura–.

2.6 Operaciones primitivas

Una vez capturado el grueso de la funcionalidad para las implementaciones objeto de estudio, se procede a la identificación de aquellas operaciones comunes a todas ellas y a las que en el contexto de este trabajo se designan bajo el término operaciones primitivas, con objeto de proporcionar una abstracción, que permita en mayor medida la operación de la unidad de protección sin necesidad de conocer los detalles intrínsecos de su arquitectura. Para ello, se toma, como ejemplo de inspiración, la interfaz de sistema operativo portátil (POSIX™) [51] [52], que declara la operación *mprotect*, tal que permite modificar la protección de acceso para las páginas –referido a un esquema de protección con soporte para la traducción de direcciones lógicas– disponibles de una tarea, dada una posición de inicio en memoria concreta. La especificación no concluye cómo se realiza la acción, pero cualquier aproximación para la provisión de servicios de sistema con arreglo al marco que define el documento, deberá definir una implementación para esta.

El principio de operación primitiva consigue que los detalles de la arquitectura pasen inadvertidos, describiendo qué efecto se desea conseguir mediante su ejecución, los valores recibidos, así como el resultado esperado tras completar su acción, por lo que si un sistema operativo desea emplear el mecanismo de protección, este deberá adherirse a la especificación que ofrece la operación primitiva. De esta manera, si la arquitectura que define la unidad de protección para un proyecto sufre modificación o difiere de la plataforma original, los cambios no se propagan al resto del sistema planteado, cuya implementación permanece inalterada. Las partes modulares que constituyen la lógica permitirían ser exportadas a otros desarrollos, siempre y cuando estos se adscriban a la declaración propuesta por las operaciones primitivas involucradas.

En cuanto a la realización de este trabajo, se impone un límite sobre la labor a realizar, tal que, la identificación de operaciones primitivas no se plantea como la definición de un estándar reglado, sino que se reduce y evalúa a una herramienta que facilita el desarrollo de la lógica asociada a la configuración e interacción de la *MPU*. Las operaciones determinan pues la funcionalidad mínima de una unidad de protección, tal que el efecto de la implementación de dichas operaciones es la de una *MPU*, la cual se ajusta a las necesidades de la aplicación y permite su uso por el sistema en cuestión. Dichas operaciones no se reducen a partes atómicas, sino que representan un conjunto de pasos a realizar, permitiendo además la combinación entre ellas.

Así, dadas las implementaciones anteriormente descritas, se consigue la identificación de las siguientes operaciones primitivas:

- Consulta de disponibilidad de regiones para nueva asignación, donde para cada implementación estudiada, se establecen un número de regiones de protección con identificador único, por ejemplo, identificador numérico o entrada en una lista de descriptores de regiones. Es posible utilizar dicho identificador, dado un momento de la ejecución –a disposición del contexto–, para el control durante la vida útil de la aplicación de aquellas regiones que prestan servicio y que por tanto no pueden ser empleadas. Además, en caso de ocupación total sería considerable notificar de este hecho al sistema. Esta operación permite su combinación con aquella destinada al intercambio de regiones, de modo

que si el número de regiones disponibles no atiende a las demandas del proceso y si las condiciones lo permiten, se liberan los recursos, si así fuere necesario.

- Asignar y/o modificar el rango de protección, tal que se asigna el valor de extensión en caso de requerir una nueva ventana o por contra, si la región ya existe, se ajusta al nuevo requerimiento. Para cualquiera de estos casos, se necesita únicamente indicar la dirección de inicio, así como el tamaño del espacio de direcciones que cubre –como se observa, algunas regiones establecen el rango como la diferencia entre las direcciones de fin e inicio, por lo que dicha dirección de fin puede obtenerse igualmente como la suma de posición inicial y el tamaño especificado–, de forma que el resultado esperado de la operación es la asignación o modificación de una ventana de forma exitosa o ante la negativa la notificación de fallo. Se propone además a modo de extensión, su integración con la anterior operación presentada –consulta–, tal que sólo se permite la asignación de un rango de protección, si existe al menos una disponible para ello.

- Asignación y/o modificación de atributos para la región, para la cual, debido a las particularidades que presentan las arquitecturas compendiadas –donde algunas unidades de protección permiten el control de atributos, los cuales determinan el comportamiento para los accesos respecto de la jerarquía de memoria y por tanto exceden la capacidad de comprobación de permisos–, se ha decidido acotar a la mera asignación de atributos asociados a los permisos para la región de protección. De esta manera, se toma como entrada una lista de permisos asociados al privilegio de ejecución –para aquellas arquitecturas donde la protección sólo se produce bajo un modo de ejecución, dicho privilegio será el mismo–, siendo el resultado esperado, en condición de éxito o fallo, el mismo que para la operación de modificación de rango.

- Habilitar o deshabilitar el mecanismo de protección de forma global, puesto que todas las arquitecturas permiten de alguna manera activar o desactivar el módulo por completo. Si el mecanismo ya se encuentra en funcionamiento o viceversa, la operación no tendrá efecto alguno. De esta forma, se permite al sistema decidir acerca de la comprobación de permisos en accesos, por ejemplo, con objeto de reducir su latencia, si una tarea con estrictos requisitos en tiempo de ejecución demanda su inhibición.

- Verificar la validez de dirección y tamaño para la ventana de protección, se precisa esta operación de forma independiente respecto a la asignación de rango de protección, por diversos motivos. Resulta de interés mantener la funcionalidad de una operación lo más sencilla posible, de forma que cambios propuestos en su especificación no incidan, en existencia de dependencias, sobre el resto, con graves consecuencias para la implementación. Además, es posible la proposición de una heurística que medie con independencia entre la creación de una tarea y la asignación de región, la cual verifica que el espacio disponible no sólo cumple con los requerimientos para la tarea, sino que además, la dirección de inicio de la ventana de protección asociada a esta –que coincide con aquella asignada para la ejecución de la tarea– y su tamaño cumplen con los requisitos impuestos por la arquitectura empleada, es decir, se permite la creación de tarea y la posterior región de protección, si y solo si se cumplen ambas condiciones. De esta manera, se eliminan operaciones innecesarias, en concreto la asignación de rango y creación de la tarea, puesto que sólo tienen lugar en caso de éxito –en caso negativo, se requiere la evaluación del impacto para este hecho, lo cual no se concibe por el momento para la definición aquí señalada–, disminuyendo entre otros aspectos la sobrecarga en ejecución.

- Creación y conservación de dominios de protección como unión de múltiples rangos, puesto que resulta de utilidad la combinación de ventanas de protección bajo un mismo punto de vista, de forma que represente el espacio disponible para un contexto de ejecución determinado. La operación deberá permitir la inclusión de nuevos rangos para un dominio ya existente o por el contrario eliminar regiones

que ya no forman parte de este, por lo que se requiere la presencia de estructuras de datos –se plantea transparente aunque la arquitectura soporte la representación mediante mecanismo *hardware*– que describan su disposición. Ante cualquier adversidad, es conveniente la notificación del hecho.

- Intercambio de dominios dado un contexto de ejecución, de forma que en lo que se refiere a la manipulación de dominios, se hace necesaria la conmutación de vistas de protección, que cada uno de ellos ofrece para un marco de ejecución determinado. Así por ejemplo, tras la planificación de una nueva tarea o en cambio de modo, el sistema selecciona la imagen correspondiente para el espacio de direcciones, de forma que las regiones de protección compartidas entre tareas se mantienen, mientras que se reemplazan aquellas necesarias para la nueva representación, notificando cualquier ocurrencia de fallo durante el proceso. La conformación del dominio por una única región, se contempla como caso particular para la operación, tal que el resultado de su acción es equivalente al intercambio individual de región, efecto conseguido al ignorar las diferencias entre una región y el dominio –como composición de regiones– para la entrada recibida.
- Determinar el origen y causa del fallo generado por acción del módulo de protección, dado que el mecanismo de protección se considera efectivo, en cuanto que no sólo evita un acceso indebido, sino que además permite conocer las condiciones que motivan la excepción. Por ello, la operación presenta el estado exacto del sistema –mediante la interacción con los registros asociados a la unidad de protección o al módulo de excepción, según las particularidades de la arquitectura–, con objeto de determinar la naturaleza del flujo anormal. Su existencia logra rutinas de servicio de interrupción del sistema lo más simple posible, dado que actúa como una rutina diferida de servicio (*DSR*), separando el paso que determina las circunstancias, de aquel bajo el cual se procede a la resolución y en consecuencia, una mayor cohesión de la lógica, de manera que las rutinas de tratamiento de fallo en acceso, residen en el mismo espacio en el que se encuentran las funciones que ofrece el sistema para manipular el módulo de protección.

2.7 MICROARQUITECTURA ESCOGIDA

Con motivo de construir un sistema *software* y permitir su operación, se demanda en última instancia un dispositivo sobre el que poder ser implantado. Es posible retrasar este requerimiento mediante una elaboración independiente, pero no puede ignorarse bajo ningún concepto, de manera que en este apartado se recoge la arquitectura para la plataforma con este respecto.

De entre todas las arquitecturas aquí propuestas, *RISC-V* suscita el mayor interés, debido a su definición bajo estándar abierto con desarrollo modular y que permite su personalización para extensión, propiedad que permitiría la combinación del mecanismo de unidad de protección con otros mecanismos hardware evaluados. No obstante, se descarta dicha opción, en cuanto que requeriría la adaptación de la lógica que conforma el núcleo bajo una *FPGA*, hecho que excede con creces la motivación de este trabajo. Por ello, la selección tan sólo contemplará el resto de las propuestas, las cuales además tienen elevada presencia en el sector de la automoción, lo que implica una elevada disponibilidad de herramientas, validadas frente a estándares con aplicación en dicho ámbito –que recogen aspectos tales como seguridad y estabilidad–, para la producción de *software*. De entre todas ellas destaca dada su implantación, tanto dentro como fuera de esta especialidad –por tanto mayor volumen de comercialización–, la arquitectura *ARMv7* y en concreto su variante *ARMv7-R*, que comparte numerosas características con aquella destinada a procesadores de alto rendimiento (*ARMv7-A*), entre las que destaca el juego de instrucciones. La elección natural en lo que refiere a la microarquitectura escogida será *Cortex-R5/R5F*.

No obstante, la elección se trata con diferencia de la más restrictiva en lo que a tamaños de región e inicio de ventana de protección respecta, hecho que impone ciertas limitaciones, las cuales serán trabajadas en lo sucesivo. Puesto que, generalmente la mayoría de operaciones sobre el espacio de direcciones requieren de alineamientos, tales restricciones ven parte de su defecto mitigado, en cuanto que las ventanas de protección se encuentran en concordancia con las particularidades que arrastran los accesos realizados, así como los objetos *software* que residen en dicho espacio. Por último, cabría destacar que el formato utilizado por la plataforma para la disposición de *bytes* (*endianess*) es *big-endian* y como se comprobará en posteriores capítulos, esto supone un contratiempo en el desarrollo del sistema aquí pergeñado.

CAPÍTULO 3

OBTENCIÓN DE REQUISITOS Y ANÁLISIS

A lo largo de este capítulo se abordará el planteamiento inicial del problema, que posteriormente confluye en un análisis del mismo, con la finalidad de profundizar en el estudio de las entidades asociadas a este, comprendiendo sus propiedades, operaciones y relaciones que manifiestan, que las conforman y las dotan de significado. Se sientan así las bases de las que depende el resto de la elaboración, tal que permiten por un lado el refinamiento de la propuesta, al dilucidar aspectos que conciernen al dominio del problema, mientras que por otro constituyen un punto de partida sobre el que recaer en existencia de cambios que afectan a la definición del alcance o al propósito del proyecto. Por tanto, el objetivo es el de condensar aquellos elementos y particularidades del problema para las que la solución deberá ofrecer servicio y de entre las que destacan las primitivas identificadas para los mecanismos de protección de memoria previamente descritas. Al tratarse de un proceso conceptual, quedan eliminadas características que refieren a aspectos particulares de la implementación y la tecnología subyacente, hecho que supone una ventaja, dado que los productos resultantes son genéricos y pueden así extrapolarse a otros proyectos de similares características (*Domain Analysis*), con arquitecturas y plataformas dispares entre sí. En este punto, la aplicación de un paradigma orientado a objetos consigue reducir la brecha semántica entre el dominio del problema y el dominio de la solución, aunque cabe destacar que este caso presenta matices, en tanto en cuanto ambos dominios incorporan conceptos correspondientes a los sistemas de información.

La organización del capítulo da comienzo con una descripción general del problema, como producto de las distintas sesiones mantenidas con el tutor y que permitirá dilucidar las características y restricciones a las que hacer frente. Tras ella resultará la tarea de obtención de requisitos, empleando una ontología común (distinguiendo entre requisitos de sistema o *SyRS* –específicos de problema en cuestión– y requisitos *software* o *SRS* –apegados a la lógica a desarrollar–) descrita en el documento *IEEE/ISO/IEC 29148:2018* [53] y que servirá como forma de adecuación y guía para su adquisición. Asimismo, el estudio de las interacciones que las entidades llevan a cabo con el sistema para satisfacer sus demandas presenta un gran interés, puesto que permiten la identificación de requisitos previmente inadvertidos. En este contexto, dicha interacción se reflejará mediante la elaboración de los denominados casos de uso. Posteriormente se introducirá la representación estructural de los conceptos que forman parte del problema mediante la implementación de un diagrama denominado modelo de dominio, al que le sucederá el desglose de la interacción entre las entidades y el sistema, mostrando el paso de mensajes y la responsabilidad asignada para cada uno. Como resultado final de estas actividades resultan los dos primeros artefactos considerados por la metodología, siendo estos los modelos *CIM* y *PIM* respectivamente.

3.1 DESCRIPCIÓN GENERAL DEL PROBLEMA

Dada la necesidad de ejecución de múltiples tareas concurrentes en el contexto de los sistemas embebidos se hace necesario garantizar la estabilidad de la aplicación y su entorno, de manera que la acción de

un proceso en condición de error no influya al resto y quede contenido a su propio ámbito. Esta capacidad puede lograrse mediante la protección de la información que las tareas utilizan o aquella que las conforman –objetos en memoria y código entre otros–, así como de los recursos que la plataforma pone a su disposición. Es por ello que se apuesta por la explotación del mecanismo de protección de memoria, incluido en sistemas empotrados basados en microcontrolador, con el fin de garantizar dicha estabilidad y sin perjuicio alguno, en mayor medida, del rendimiento para la ejecución. No obstante, este mecanismo carece de relevancia como ente autónomo, por lo resultará de interés la concepción de un sistema que englobe en su marco toda aquella funcionalidad que posibilite la interacción y configuración del elemento de protección. Dicho sistema también aglutinará otros servicios, propios de un *kernel*, como la planificación de tareas, gestión de recursos y comunicación entre distintas entidades. En definitiva, este sistema habilita el estudio de este tipo de técnicas de protección de memoria y sirve como prototipo para el futuro desarrollo de aplicaciones en el contexto de estos sistemas de información y en particular para aquellos basados en la arquitectura *ARM*®. Cabe señalar que este *kernel* mínimo se plantea como una prueba de concepto, en cuanto que se delimitan sus características, por lo que se deberán tener en cuenta convenientemente tales circunstancias durante la elaboración del producto.

En primer lugar se plantea una planificación de tareas con propiedades estáticas y basada en prioridad. Con respecto a estática se refiere al apriorismo de su existencia, es decir, se advierte el número total de procesos que requieren ejecución de antemano, así como sus atributos tales como el tamaño que ocupan en memoria, su posición concreta en el espacio de direcciones, los servicios que requerirán para completar su función, su nivel de prioridad y el número de regiones de protección asignado para cada uno –supeditado al máximo disponible dispuesto por la arquitectura a emplear–. Tal y como se enuncia, la asignación de ventanas de protección presentará el mismo carácter estático, de manera que no se contempla una heurística para la creación de regiones ni tareas de forma dinámica, por lo que tampoco se requiere de la existencia de un localizador capaz de resolver las direcciones asociadas a un proceso en tiempo de ejecución. Cabe precisar que una prioridad éstática –que no inmutable– garantizará un mismo orden en planificación de tarea para ejecuciones cualesquiera, de modo que el flujo seguido es previsible ante la ausencia de cualquier defecto. Las tareas podrán ver interrumpida su acción –tras un tiempo previamente establecido y conocido como (*quantum*)– aún sin haber completado sus operaciones, por lo que será necesaria la presencia de un mecanismo de interrupciones programáticas, o por contra estas pueden acabar con su propia acción antes siquiera de agotar su tiempo asignado, de forma que ceden el control de los recursos voluntariamente, permitiendo así la planificación de una nueva tarea. Es conveniente aclarar que tanto los atributos y permisos que condicionan las operaciones de acceso pueden sufrir alteraciones en circunstancias particulares, por ejemplo tras la detección de un fallo, es decir un flujo no previsto para la lógica. Asimismo podrán asociarse múltiples ventanas de protección a una tarea concreta, hecho posible mediante la mediación del dominio, el cual es intercambiado en cambio de contexto durante la planificación. No se especifica la implementación de estos dominios, pero resulta evidente que al menos involucra la agrupación de regiones –si la plataforma ofrece soporte también podrán emplearse subregiones como caso particular–. Entre otras funciones que deberán ser provistas se demanda la inhibición del mecanismo de protección en su conjunto o de regiones individuales.

Junto a la labor de protección del espacio deberá asegurarse la capacidad de recuperación y disponibilidad del sistema frente a fallos –cabe destacar la existencia de un amplio catálogo de fallos que no serán tratados en este trabajo– en operación de acceso, una vez estos han tenido lugar. El sistema dispondrá de la correspondiente funcionalidad que permita recabar información como la identificación del origen y causa del evento señalado por la unidad de protección de memoria. Si se confirma su presencia, entonces se de-

berán establecer procedimientos para el tratamiento de este, de modo que se garantiza un flujo de ejecución conocido y determinado tras la recuperación. En concreto, si el fallo no modifica el estado previsto de la ejecución ni de su entorno, entonces la tarea responsable, siempre que pueda trazarse una correspondencia, es eliminada de la planificación, de forma que la ejecución deberá continuar sin la presencia de dicha tarea hasta un reinicio, por lo que es indispensable notificar del suceso. Si por el contrario sí ocasiona un cambio en el estado previsto, entonces se tiene la obligación de revertir los cambios producidos, en caso de que estos sean recuperables, mientras que ante la imposibilidad de este hecho se deberá forzar un reinicio de la plataforma tras notificar la acción. Además, se deberán proveer funciones de limpieza del entorno de manera previa a la acción de reinicio de la plataforma, ya que se deberá garantizar un estado confiable tras el arranque, evitando así posibles fallos inesperados al retomar la ejecución tras el reinicio.El sistema también podrá beneficiarse de las denominadas dimensiones de ejecución –tantas como se encuentren presentes la arquitectura– con objeto de garantizar la estabilidad de la plataforma. Generalmente pueden distinguirse dos tipos de modos, donde bajo la primera, denominada como dimensión o espacio del *kernel*, se enmarcan los procesos y servicios relativos al sistema, entre los que se encuentra el mecanismo capaz de mediar con la unidad de protección, mientras que la segunda o dimensión de usuario alberga las tareas que conforman la aplicación. Establecida esta posible división se hace hincapié en la siguiente característica, tal que el sistema no permitirá ningún acceso directo desde el denominado espacio de usuario a aquellos recursos controlados de manera exclusiva por el *kernel*. Para ello el sistema pondrá a disposición de las tareas un nexo de comunicación confiable, conocido como llamadas a sistema, de forma que cualquier petición es mediada y regulada. El sistema podrá disponer por tanto de dichos modos en sintonía con la *MPU*, la cual permite el control individual de dimensión para cada una de las regiones disponibles.

Por último y con motivo de evaluar la efectividad de este mecanismo, el prototipo bajo el que se integra la técnica de protección deberá garantizar la ejecución concurrente de al menos dos tareas, correspondientes ambas a la dimensión del espacio de usuario. La función que cumplen estas tareas es la de generar una anomalía en el flujo de ejecución, la cual se presupone prevista y conocida de antemano. No obstante, no se establece ningún tipo de definición para dicha anomalía, sino que únicamente se demanda la visibilidad tanto de la capacidad en la labor de protección como de los recursos que ofrece la *MPU*. En la misma medida, tampoco se especifíca una función concreta para ninguna de estas tareas, aunque estas sí deberán dejar constancia de su ejecución de algún modo, por ejemplo, mediante una interacción visual, como el encendido y apagado de un led de la plataforma o mediante el envío de mensajes a través de una interfaz de comunicación. Cabe añadir la necesidad de un estado de reposo, cuya finalidad es la de reducir el consumo energético de la plataforma dada una situación de planificación ociosa.

3.2 Obtención de requisitos de sistema (*SySR*) y requisitos software (*SRS*)

La especificación de características que definen el comportamiento del sistema y las particularidades que afectan a este resultan de gran importancia, dado que constituyen un pilar fundamental sobre el cual se apoyará la totalidad del proyecto. Existen numerosas técnicas que facilitan la extracción de tales características o requisitos atendiendo a aquellos aspectos que permiten su clasificación. En lo que compete a este trabajo, el empleo del estándar *IEEE/ISO/IEC 29148:2018* [53], el cual establece una taxonomía entre requisitos de sistema y *software*, ayudará a estipular aquellos disponibles para este trabajo.Del mismo modo, la condición de prototipo, cuyo objetivo es la evaluación de las bondades de la unidad de protección como mera demostración de posibilidad y capacidad, resulta determinante para la selección llevada a cabo. Tal es así que no se contempla la inclusión de requisitos referidos a condiciones exógenas al no considerarse estos

como centrales, aquellos que refieren al entorno, en concreto los denominados requisitos de condiciones del entorno, así como tampoco una serie de requisitos propios de proyectos con aplicación real de entre los que por ejemplo destacan los denominados requisitos asociados con el rendimiento. Así pues, la lista de los requisitos disponibles para este trabajo se resume a continuación.

3.2.1 Requisitos funcionales

RF001 – Estado de ejecución confiable y conocido

El sistema deberá garantizar un estado conocido, dependiente de la arquitectura, tras el arranque de la plataforma, tal que se permita albergar la ejecución de las tareas que constituyen la aplicación.

RF002 – Separación en modos de ejecución

El sistema garantizará la separación de la ejecución en al menos dos dimensiones –privilegiado y usuario–.

RF003 – Comprobación de estado de la plataforma

El sistema proveerá de un mecanismo para la consulta y control del estado ejecución de la plataforma.

RF004 – Estados de ejecución

El sistema permitirá discernir entre los diferentes estados de ejecución en los que se encuentra una tarea, esto es, inactiva –o en espera–, activa y no disponible.

RF005 – Planificación de tareas

El sistema proporcionará un mecanismo para la planificación de tareas en prioridad.

RF006 – Límite de ejecución

El sistema deberá garantizar la finitud de las tareas.

RF007 – Cesión de tiempo

El sistema permitirá a una tarea ceder su tiempo de ejecución para la planificación.

RF008 – Conservación y cambio de contexto

El sistema conservará la información de ejecución y contexto para cada una de las tareas disponibles y permitirá su intercambio tras la planificación.

RF009 – Coexistencia de tareas

El sistema soportará la coexistencia de al menos dos procesos.

RF010 – Llamadas al sistema

El sistema proveerá a las tareas de un mecanismo para la invocación de servicios que estas requieren.

RF011 – Modo de bajo consumo

El sistema ofrecerá un estado de reposo en ausencia de tareas para su planificación.

RF012 – Asignación de ventanas de protección

El sistema permitirá la creación, configuración –atributos, permisos, extensión y dirección base– y modificación de regiones de protección.

RF013 – Agregación en dominios de protección

El sistema agrupará regiones de protección –en existencia de dos o más– para un contexto de ejecución común bajo los denominados dominios de protección.

RF014 – Restaurar dominios de protección

El sistema permitirá el intercambio de dominios de protección.

RF015 – Detección de fallo

El sistema facilitará la detección de origen y causa en fallo.

RF016 – Gestión y tratamiento de fallo

El sistema deberá gestionar y tratar los eventos de fallo.

RF017 – Notificación de condición de error

El sistema notificará la detección ante cualquier tipo de fallo.

RF018 – Continuación en condición de fallo recuperable

El sistema deberá garantizar la continuidad para la ejecución, así como delimitará un estado a alcanzar en caso de fallo recuperable.

RF019 – Control atómico en condición de fallo

El sistema habilitará o deshabilitará en condición de fallo un rango de protección bajo un contexto de ejecución.

RF020 – Control global en condición de fallo o cambio de contexto

El sistema permitirá habilitar o deshabilitar por completo el mecanismo de protección de memoria durante el tratamiento de fallo o en cambio de contexto.

RF021 – Acceso a información específica del mecanismo de protección

El sistema deberá ser capaz de consultar el número total de regiones de protección para la arquitectura empleada.

RF022 – Control de comunicaciones

El sistema garantizará la imposibilidad para la comunicación indebida o no prevista entre contextos de ejecución o entre tareas.

RF023 – Interfaz para la comunicación de información

El sistema proveerá a las tareas de una interfaz basada en comunicación serie, así como de entrada-s/salidas genéricas, para la captura de información.

3.2.2 Requisitos no funcionales

RNF001 – Lenguaje de programación C

Se empleará el lenguaje de programación C como aquel destinado a la realización del prototipo.

RNF002 – Valor de *quantum*

El valor de cuanto de ejecución será de 10ms.

RNF003 – Soporte para arquitecturas ARMv7-R

La arquitectura ARMv7-R será aquella escogida para albergar al prototipo.

RNF004 – Comunicación mediante led

Al menos un posible canal para la comunicación deberá emplear una salida génerica en cuyo extremo se encuentra un led de notificaciones.

RNF005 – Comunicación mediante interfaz serie UART

Al menos un posible canal para la comunicación deberá emplear la interfaz UART.

RNF006 – Generación de interrupciones de tiempo real

La finitud de la ejecución para las tareas se establecerá mediante un mecanismo de generación de interrupciones de tiempo real.

3.3 Casos de uso

La protección de memoria constituye uno de los servicios que deberá ofrecer el prototipo planteado. Con la finalidad de demostrar sus capacidades y así justificar su concepción, este mecanismo deberá permitir la relación con su entorno, así como con las aplicaciones que alberga. Resulta pues interesante capturar los requisitos correspondientes a dicha interacción con entidades externas, junto a las condiciones y capacidades a las que debe adecuarse su funcionalidad. De esta manera, se la búsqueda de una técnica capaz de incorporar tal información de forma concisa y precisa, tal que sirva como un punto de partida para la realización del producto. El mejor candidato para esta labor puede encontrarse en los denominados casos de uso, herramientas de utilidad a la hora de comprender aquello que se pretende construir. Estos consiguen delimitar las fronteras y el comportamiento del sistema, en tanto en cuanto se centran únicamente en los problemas a los que ofrecer solución mediante la elaboración del *software*, de modo que no tienen en su haber aquello ajeno al propio sistema. Así pues y en lo sucesivo se pretende dar cuerpo a los casos de uso correspondientes al proyecto, tomando [54] como referencia para cada uno de los subapartados.

3.3.1 Catálogo de actores

Tal y como se precisa, los casos de uso resultan de utilidad para la definición de los límites del sistema, efecto que se consigue de manera transversal al identificar a los actores que participan en cada uno de ellos. Pueden distinguirse dos categorías de actores en función de su participación en la interacción, lo cual además determina el rol que tomará el sujeto real que representa y cuyas diferencias se contraponen a continuación:

- Actor principal, tal que se entiende como una entidad externa, la cual obtiene valor al emplear el sistema informático puesto a su disposición para la consecución de sus objetivos. Por tanto, son aquellos que dotan de sentido a la creación del sistema y que generalmente dan inicio al caso de uso mediante su acción (aunque no siempre resulta así).

- Actor de soporte, cuya función consiste en proporcionar servicios de apoyo al sistema, que resultan esenciales para el correcto desarrollo del caso de uso. No necesariamente requieren de una representación explícita, esto es, su incorporación tiene sentido si concretan condiciones no eludibles para el flujo, de modo que este no puede continuar sin su existencia. Al igual que los actores principales se benefician de la correcta ejecución de estos, es decir, cuando no se encuentra un flujo anormal y se completan tal y como son concebidos, pero a diferencia de los anteriores, no demandan obligatoriamente la satisfacción de sus fines.

Una vez, pues, establecidas las categorías posibles para este subapartado se procede a la identificación de actores relativos al prototipo elaborado, tal que pueden observarse los presentes:

- Tarea, como un sujeto abstracto ajeno al propio prototipo, que se enmarca en la categoría de actor principal, en tanto en cuanto demanda y requiere de servicios para completar su acción. Este actor presenta todas aquellas propiedades que constituyen a las tareas que interactúan con el sistema. No obstante, dicha interacción no se lleva a cabo de forma directa por el actor, sino que lo hacen sus posibles estados dinámicos, recogidos en la figura 3.1, que determinan su comportamiento y por ende sus objetivos, así como los puntos de interacción para cada uno de ellos.

- Tarea activa, como uno de los posibles estados para el actor tarea. La interacción con el sistema permitirá cubrir sus necesidades durante la ejecución, puesto que de ningún otro modo podría llevar a cabo su acción.

- Tarea en espera, que demanda la capacidad de planificación del sistema para lograr sus metas. No compete a este actor el inicio de ningún flujo, aunque resulta el principal beneficiado de acciones directas o indirectas del sistema. Tras los servicios del sistema este cambia de estado, erigiéndose como nueva tarea activa.

- Sistema anfitrión, tal que registra una particularidad respecto de su clasificación como actor que atañe bien a la consideración como un sistema distribuido, en donde las decisiones no dependen de un punto común, o por el contrario centralizado. La entidad se entiende como receptora de eventos para el primero de los casos –por ejemplo en existencia de error–, de modo que su comportamiento se enmarca en los denominados actores principales. Sin embargo, si la continuación del flujo del caso de uso depende de información provista por dicho sujeto a modo de arbitraje, entonces este se interpreta como un posible actor de soporte.

- Temporizador, donde su existencia como actor de soporte resulta necesaria, dado que provee del evento que da pie a la planificación de manera periódica. Su existencia como sujeto escapa al control del prototipo, que establece la comunicación para la configuración de este mecanismo, generalmente provisto por la plataforma *hardware* subyacente.

- Unidad de protección de memoria, siendo este el mecanismo que representa el tema central tratado en este trabajo y de igual modo que el temporizador, este corresponde a los denominados actores de soporte. Su funcionamiento excede de nuevo las competencias del sistema, en cuanto que es provisto por la plataforma, aunque no obstante ambos conocen de su existencia, dado que se relacionan entre sí para determinar la configuración de regiones de protección. Este actor se encargará, entre otros aspectos, de detectar un acceso indebido –siempre y cuando al menos una ventana de protección sea válida– y de generar la correspondiente señal de condición de error a capturar por el sistema.

- Mecanismo de escalado de privilegios, como actor de soporte y por tanto un servicio extrínseco al sistema, sobre el que delegar la acción de cambio de modo para así obtener los permisos necesarios dado un contexto de ejecución.

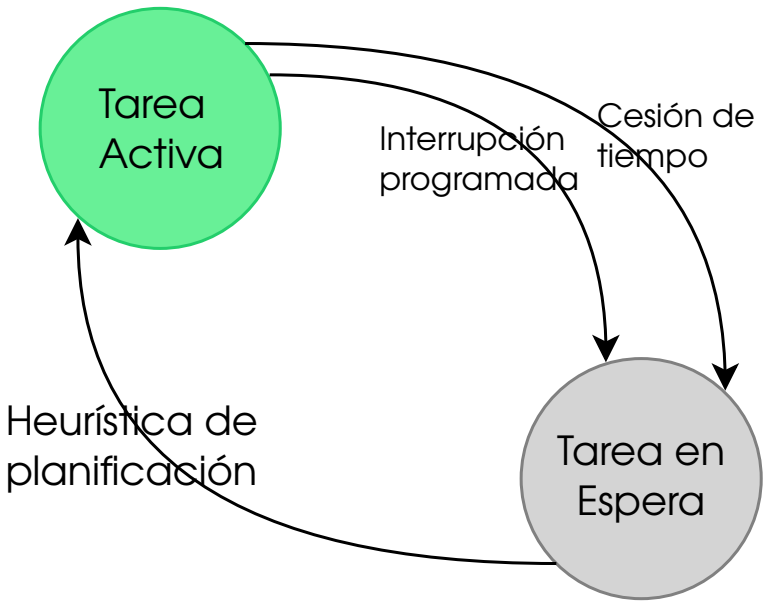

FIGURA 3.1: Posibles estados en los que puede encontrarse una tarea

3.3.2 Especificación de casos de uso

Provistos aquellos actores participantes se procede a la identificación y posterior descripción de los casos de uso. Para ello se empleará una forma narrativa, en tanto en cuanto posibilita una posterior definición completa a partir de ella, si así fuese necesario

UC:0001 – Llamada a servicio del sistema

Descripción

Este caso de uso describe el proceso para el que un actor **Tarea activa** demanda los servicios bajo llamada, que le ofrece como apoyo el sistema para continuar con su acción.

Extensión

No aplica.

Precondiciones

○ Al menos se encuentra una tarea en ejecución.

○ La plataforma permanece en el modo usuario y bajo un estado conocido y confiable.

Escenario básico

{Solicitud de servicio}

1. El caso de uso da comienzo cuando el actor **Tarea activa** solicita un servicio de sistema para continuar con su ejecución.

{Validación de solicitud}

2. El sistema valida la petición recibida de acuerdo a la definición del servicio:

2.1. Si se requiere de permisos especiales para el servicio solicitado, entonces el sistema realiza el subflujo **Escalado de privilegios**.

2.2 En caso contrario, el sistema continúa la acción en el mismo modo en el que lo hace la tarea y por tanto usuario.

{Invocación de servicio}

3. El sistema determina el código concreto de operación asociado al servicio solicitado y despacha el correspondiente servicio.

{Fin del caso de uso}

6. El caso de uso finaliza.

Flujos alternativos

Para **{Solicitud de servicio}**, si la comprobación no coincide con la definición establecida para el servicio,

1. El sistema informa al actor **Tarea activa** del incumplimiento de formato, indicando aquellos aspectos no válidos.
2. El sistema despliega una lista con las posibles entradas soportadas por el servicio.

Para **{Invocación de servicio}**, si la acción del servicio no se completa o lo hace inadecuadamente,

1. El sistema captura el tipo de evento de error software producido.
2. El sistema informa a la tarea de la ocurrencia de este evento.

Subflujos

S1 – Escalado de privilegios

1. El sistema comprueba si el tipo de operación de servicio dispone de los permisos adecuados.
2. Si se carece de tales privilegios, entonces el sistema se comunica con el actor de soporte **Mecanismo de escalado** con este fin.
3. El mecanismo de soporte modifica el estado de la plataforma, otorgando así las capacidades requeridas.
4. Se prosigue en el siguiente paso del caso de uso.

Postcondiciones

○ Se completa el servicio solicitado y la tarea puede continuar con su acción.

UC:0002 – Tratar fallo en acceso indebido

Descripción

Este caso de uso contempla el tratamiento de la interacción que el actor **Unidad de protección de memoria** realiza con el sistema cuando este notifica la detección de un acceso no previsto sobre una dirección del espacio protegida mediante una región activa.

Extensión

No aplica.

Precondiciones

○ Se parte de un estado de ejecución para el que no existen eventos pendientes por tratar y es posible recibir condiciones de error.

Escenario básico

{Capturar evento de error}

1. El sistema captura una señal de error generada por el actor de soporte **Unidad de protección de memoria**, en el momento que detecta una acceso indebido por el actor **Tarea activa**.

{Inquirir estado de condición}

2. El sistema solicita al actor de soporte **Unidad de protección de memoria** toda la información asociada a la condición de error, entre la que se encuentran datos relativos a la tarea propietaria de la región y la tarea activa.

{Manejar evento de fallo}

3. El sistema lleva a cabo el diagnóstico y tratamiento para el fallo:
   3.1. Si el error detectado es la consecuencia de un estado inestable e irrecuperable de la plataforma, entonces el caso de uso continúa en el paso 5.
   3.2 Si se responsabiliza de tal acción de fallo al actor **Tarea activa** y no es posible asegurar su continuidad, entonces el sistema señala su estado como no planificable. El caso de uso continúa en el siguiente paso.
   3.3 En caso contrario se recupera el estado del actor **Tarea activa** previo a la condición, sobreescribiendo su valor actual, que se considera de riesgo. El caso de uso continúa en el paso siguiente.

{Confirmar tratamiento}

4. El sistema informa al actor de soporte **Unidad de protección de memoria** que el fallo ha sido debidamente tratado.

{Notificación de suceso}

5. El sistema notifica de la ocurrencia de este suceso al actor principal **Sistema anfitrión**, así como de la solución tomada y el estado actual de la plataforma. **{Fin del caso de uso}**
6. El caso de uso finaliza.

Flujos alternativos

En **Manejar evento de fallo** y si no es posible el reinicio:

1. El sistema comunica de la gravedad de la situación al sistema anfitrión.
2. El sistema permanece a la espera de una acción externa provista por el actor **Sistema anfitrión**

Subflujos

No aplica.

Postcondiciones

○ El actor **Tarea activa** recupera su estado anterior a la ocurrencia del evento.
○ El sistema anfitrión queda notificado de la existencia del error producido.
○ No quedan eventos pendientes por tratar previo paso a la reanudación de la ejecución, en el punto concreto en el que se produce el acceso que generó el evento.
○ La plataforma recupera un estado viable para la ejecución.

UC:0003 – Planificar tareas

Descripción

El siguiente caso de uso expresa el tratamiento y control de recursos en existencia de tareas concurrentes, mediante el uso de una heurística, que determina la siguiente tarea en ejecución. Su acción tiene lugar bien como un servicio en la cesión de tiempo, tras agotar el cuanto de tiempo asignado para un proceso o como posible solución bajo una condición de error en acceso indebido.

Extensión

Extiende al caso de uso **Llamada a servicio del sistema** en el punto **Invocación de servicio**.
Extiende al caso de uso **Tratar fallo en acceso indebido** en el punto **Manejar evento de fallo**.

Precondiciones

○ Se señala la existencia de un evento programado.
○ Se solicita la acción del servicio.
○ La plataforma se encuentra en modo privilegiado

Escenario básico

{Condiciones para el inicio}

1. El caso de uso puede tener lugar dadas varias condiciones:
   1.1. El caso de uso extiende una invocación a servicio de sistema por parte de la **Tarea activa**, por lo que el sistema atiende la petición siguiendo el flujo aquí expuesto.
   1.2. El actor **Temporizador** avisa de la llegada de una señal de interrupcion programada, de modo que el sistema inicia este flujo.
   1.3. Como posibilidad de tratamiento de un fallo de acceso, de manera que el sistema plantea la planificación de una nueva tarea.

{Conservación de contexto}

2. El sistema preserva el contexto de ejecución:
   2.1. Si se proporciona el servicio como efecto del tratamiento de fallo, entonces se continúa en el paso siguiente.
   2.2. En caso contrario, se conserva el estado del actor **Tarea activa**.

{Preparación de tiempo de cuanto}

3. El sistema realiza el flujo descrito en el caso de uso **Manejar interrupción periódica**.

{Decisión de candidato}

4. El sistema aplica una heurística sobre todas las tareas que interactúan con este y se encuentran disponibles para la planificación, con objeto de obtener un candidato

{Restaurar dominio de protección}

5. El sistema continúa en el caso de uso **Restaurar ventanas de protección**.

{Reinicio del temporizador}

6. El sistema realiza el subflujo **Activación de interrupción periódica**. **{Delegar la ejecución}**
7. El sistema devuelve el control de la ejecución a la correspondiente tarea:
   7.1. Si una nueva tarea resulta de la planificación, entonces la transferencia se produce sobre esta y no necesariamente a aquella que solicita el servicio.
   7.2. En cualquier otro caso se retorna a aquella que inicia la petición. **{Fin del caso de uso}**
8. El caso de uso finaliza.

Flujos alternativos

Para **Decisión de candidato**:

1. Si se advierte imposible la planificación, entonces se notifica de este hecho al actor **Sistema anfitrión**.
2. El sistema queda a la espera de una decisión por parte del **Sistema anfitrión**.

Para **Reinicio del temporizador**:

1. Si no es posible restaurar el funcionamiento del mecanismo de interrupciones, entonces la ejecución no puede continuar y se comunica de este hecho al **Sistema anfitrión**.
2. El sistema queda a la espera de una decisión por parte del **Sistema anfitrión**.

Subflujos

S1 – Activación de interrupción periódica

1. El sistema solicita al actor de soporte **Temporizador** que reanude el mecanismo de interrupciones programadas.
2. El sistema queda a la espera de la confirmación del paso anterior.
3. El sistema comprueba la respuesta:

3.1 Si aún no se encuentra listo entonces vuelve al paso 2 de este subflujo.

3.2 Si la respuesta es afirmativa, entonces prosigue en el siguiente paso del caso de uso.

Postcondiciones

- Se provee una nueva entidad como actor **Tarea activa**, que disfruta de los recursos de la plataforma.

UC:0004 – Manejar interrupción periódica

Descripción

Este caso de uso se incluye en el caso de uso **Planificar tareas** y permite tratar adecuadamente la configuración de una nueva interrupción programada durante la planificación, mediante la interacción con el actor de soporte **Temporizador**.

Extensión

No aplica.

Precondiciones

- La plataforma se encuentra en un estado de ejecución conocido y confiable.
- El control de la ejecución corresponde al sistema.
- El sistema ha sido notificado de un evento de interrupción programada que indica el agotamiento del cuanto.
- El sistema se encuentra en el proceso de planificación.

Escenario básico

{Inhibición de captura de señales}

1. El sistema procede al enmascaramiento de señales programáticas, de modo que no resulta interrumpido en los sucesivos pasos.
2. El sistema realiza el subflujo **Parado de temporizador**

{Comprobación de eventos pendientes}

2. El sistema procede a comprobar si existen eventos pendientes:
   2.1. Si existen eventos pendientes, el sistema limpia la notificación recibida con objeto de permitir futuras interrupciones programadas.
   2.2 En caso contrario continúa en el siguiente paso.

{Recargar el valor de cuanto}

3. El sistema realiza el subflujo **Reinicio de temporizador**

{Reactivación de captura de señales}

4. El sistema restaura la captura de eventos programáticos, dado que el temporizador se asume aún como inactivo.

{Fin del caso de uso}

5. El caso de uso aquí presentado finaliza y el sistema retoma el siguiente paso correspondiente al flujo del caso de uso **Planificar tareas**.

Flujos alternativos

Para **Inhibición de captura de señales**, **Recargar el valor de cuanto** y **Activar unidad de protección**:

1. Si ocurre un evento no previsto durante la comunicación con el actor **Temporizador** o este no completa correctamente las solicitudes, entonces el sistema notifica al actor **Sistema anfitrión**.
2. El sistema queda a la espera de la toma de decisión por parte del **Sistema anfitrión**.

Subflujos

S1 – Parado de temporizador

1. El sistema solicita la parada del mecanismo de interrupciones al actor de soporte **Temporizador**.
2. El sistema queda a la espera de la confirmación del paso anterior.
3. El sistema comprueba la respuesta:
   3.1 Si la respuesta aún no se encuentra disponible, entonces vuelve al paso 2 de este subflujo.
   3.2 Si la respuesta es afirmativa, entonces prosigue en el siguiente paso del caso de uso.

S2 – Reinicio de temporizador

1. El sistema suministra al actor de soporte **Temporizador** el valor de cuanto para su programación.
2. El sistema queda a la espera de la confirmación del paso anterior.
3. El sistema comprueba la respuesta:

3.1 Si la respuesta aún no se encuentra disponible, entonces vuelve al paso 2 de este subflujo.

3.2 Si la respuesta es afirmativa, entonces prosigue en el siguiente paso del caso de uso.

Postcondiciones

○ La siguiente interrupción resulta correctamente configurada, esto es se ajusta a las necesidades de tiempo de ejecución para las tareas.

○ Se retoma la búsqueda de una nueva tarea candidata para la transferencia de control.

UC:0005 – Restaurar ventanas de protección

Descripción

Este caso de uso habilita el dominio requerido por la tarea escogida, como consecuencia del cambio de contexto durante la planificación.

Extensión

No aplica.

Precondiciones

- El contexto del actor **Tarea activa** se encuentra convenientemente intercambiado.
- Se encuentra presene en la plataforma al menos una unidad de protección de memoria.

Escenario básico

{Deshabilitar unidad de protección}

1. El sistema realiza el subflujo **Inhibir unidad de protección**.

{Intercambio de regiones}

2. El sistema realiza el subflujo **Intercambiar regiones**

{Activar unidad de protección}

3. El sistema realiza el subflujo **Relanzar unidad de protección**.

{Fin del caso de uso}

4. El caso de uso finaliza y el sistema continúa en el siguiente paso previsto para el flujo de **Planificar tareas**.

Flujos alternativos

Para **Deshabilitar unidad de protección**, **Intercambio de regiones** y **Activar unidad de protección**:

1. Si ocurre un evento no previsto durante la comunicación con el actor **Unidad de protección de memoria** o este no completa correctamente las solicitudes, entonces el sistema notifica al actor **Sistema anfitrión**.
2. El sistema queda a la espera de la toma de decisión por parte del **Sistema anfitrión**.

Subflujos

S1 – Inhibir unidad de protección

1. El sistema solicita al actor de soporte **Unidad de protección de memoria** la inhibición del mecanismo.
2. El sistema permanece a la espera del cambio de estado.
3. El sistema demanda al actor de soporte **Unidad de protección de memoria** el estado de la unidad de protección:
   - 3.1. Si la comprobación es positiva, esto es se encuentra deshabilitada, entonces se continúa en el paso 4. del subflujo
   - 3.2. En caso contrario se vuelve de nuevo al paso 2.
4. Se continúa en el siguiente paso del caso de uso.

S2 – Intercambiar regiones

1. El sistema comprueba el dominio asociado al actor candidato **Tarea en espera**.
2. El sistema provee al actor de soporte **Unidad de protección de memoria** con la información correspondiente a cada una de las regiones del dominio.

S3 – Activar unidad de protección

1. El sistema demanda al actor de soporte **Unidad de protección de memoria** habilitar la unidad de protección.
2. El sistema permanece a la espera del cambio de estado.

3. El sistema requiere el estado de la unidad de protección al actor de soporte **Unidad de protección de memoria**:
   3.1. Si la comprobación es positiva, esto es se relanza su funcionamiento y se prosigue en el paso 4. del subflujo
   3.2. En caso contrario se vuelve de nuevo al paso 2.
4. Se continúa en el siguiente paso del caso de uso.

Postcondiciones

○ El dominio asociado al actor **Tarea activa** es reemplazado por el actor candidato para la planificación.

3.3.3 Representación de casos de uso

La caracterización de los casos de uso especificados bajo un diagrama permite simplificar su representación y pone el foco por un lado en delimitar todo aquello que pertenece al sistema o es ajeno a él, y por otro en las relaciones existentes para las entidades que hacen uso de este, así como entre los distintos casos de uso. Cabe destacar que en el ámbito de los sistemas empotrados dicha representación sufre de ciertas carencias, ya que la sintaxis ofrecida por la especificación *UML* no permite incorporar características tales como concurrencia, generación de eventos e interrupciones entre otras. Además, en este tipo de sistemas es frecuente la existencia de flujos que tienen lugar como resultado de un evento –por ejemplo una interrupción– y no por la interacción directa de una entidad. Para hacer frente a estas limitaciones se incorporan extensiones mediante el uso de estereotipos, tal y como recoge [55], para el conjunto de los actores que participan. Así pues, el estereotipo evento (*event*) refiere a un rol tal que su acción indica el comienzo del caso de uso y por otro, el estereotipo interfaz, que implica una vía de comunicación con otro dispositivo, usuario o elemento del entorno, sin la necesidad de especificar detalles concretos de la comunicación.

Para la totalidad el diagrama se establecerán las siguientes relaciones, a saber:

- Relación de especialización, para la que tal y como se ha mencionado de manera previa indica características comúnes entre los distintos estados posibles para una tarea.
- Relación de extensión entendida como un flujo opcional y en determinadas circunstancias excepcional, tal que se inserta en otro caso de uso. Suele caracterizarse en forma de condición en un punto de extensión público para el caso de uso extendido, de forma que si se cumple una casuística, entonces tiene lugar el flujo alternativo que amplica las capacidades del escenario básico.
- Relación de inclusión, que a diferencia de la anterior, el flujo agregado resulta imprescidible para considerar completo el caso de uso base. Su aparición sólo guarda sentido cuando un flujo es común a más de un caso de uso, de manera que este puede ser reutilizado sin restricción de uso.
- Relación de uso la cual resulta, dada su especificidad, un tanto especial respecto de las anteriores, en cuanto que permite añadir mayor información. Vendría a indicar la dependencia que tiene el flujo principal para completar su acción, de manera que delega en el caso de uso objetivo.

A continuación se ofrecen algunos apuntes relativos al diagrama 3.2:

- Tal y como se observa, se establecen numerosas relaciones entre los distintos actores y los casos de uso presentado. Este criterio obedece a la alta participación, bien directa o indirectamente de los actores, dadas las características del problema, en donde actuadores envían señales, comunicando así el inicio de un caso de uso, hecho que se refleja mediante una línea sólida con terminación en

punta de flecha, pero que asimismo quedan a expensas de recibir información y se sirven de los flujos establecidos para cumplir sus objetivos.

- La decisión de establecer el caso de uso **Planificar tareas** como una extensión de llamadas a servicios obedece a la distinción entre puntos de entrada para las bondades del sistema. De esta manera se pretende reflejar el flujo de interacción para el actor **Tarea activa** en cesión de tiempo, donde el sistema se encarga tanto de la mediación como de la planificación, así como aquel que obedece a la señal de interrupción periódica, que el sistema procesa de forma directa.
- La separación entre **Manejar interrupción periódica** y **Planificar tareas** se encuentra motivada por la separación de intereses. Así pues, dado un futuro desarrollo donde el tipo de interrupciones periódicas puede aumentar y que propicia un incrementa del número de rutinas de servicio asociada, esta división permite reutilizar el flujo que lleva a cabo la interacción con el actor **Temporizador** con independencia de dichas rutinas.
- La relación de uso establecida entre **Planificar tareas** y **Restaurar ventanas de protección** hace énfasis en la temática central para el trabajo. Durante la planificación no sólo se concede a la tarea candidata acceso a los recursos de la plataforma, sino que además se delimita el dominio y por tanto el rango del espacio visible para ella.

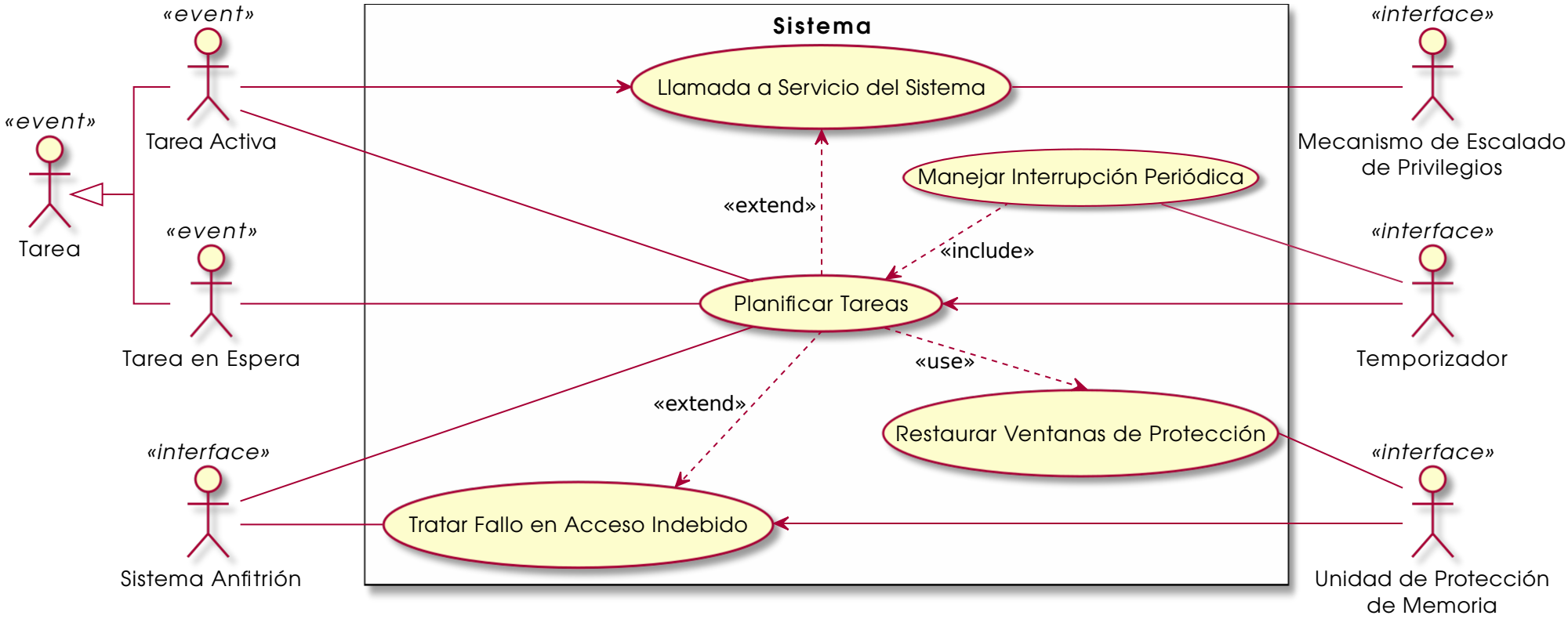

FIGURA 3.2: Casos de uso identificados

3.4 Modelo de Dominio

Con objeto de manejar la complejidad asociada al problema se emplea la realización de un modelo de dominio, mediante un diagrama de clases, tal que permite capturar de forma abstracta todos aquellos conceptos que forman parte del dominio del problema. La obtención de estos conceptos resulta de las actividades previas y sus representaciones pretenden reflejar las propiedades, así los comportamientos que los definen. Este modelo se considerará un agregado del modelo *PIM*, que resulta el producto del análisis.

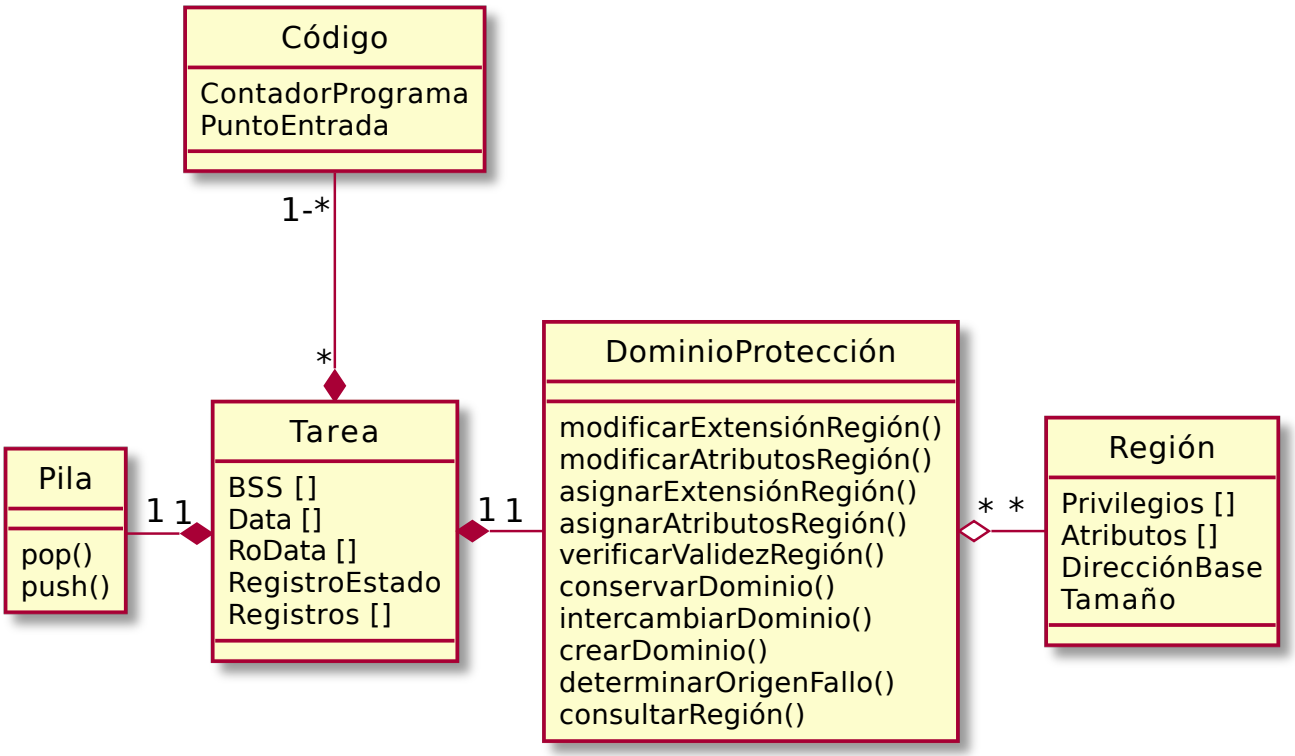

Figura 3.3: Modelo de dominio

La figura 3.3 presenta la composición de las tareas para las que el sistema ofrece sus servicios. Estas tareas se perciben como un agregado de secciones de código –que conforman un único componente para el que se deberá especificar tanto la dirección de inicio, así como la última dirección ejecutada–, datos y un conjunto de registros que conforman su estado. Además, requieren de un espacio de pila propio para lograr su acción, dadas las restricciones que presentan las arquitecturas subyacentes a nivel de registros de unidad de procesamiento. Puesto que cada tarea requiere de su dominio de protección, que corresponde al espacio visible por esta durante su ejecución e inaccesible –excepto fragmentos compartidos– por ninguna otra tarea, se debe adjuntar como parte de esta composición el conepto en referencia al dominio único e intransferible. El dominio de protección se entiende como la agregación de regiones, de forma que si no existe ninguna región asociada a este, entonces la tarea dispone de la totalidad del espacio. Tal y como se observa, el dominio refleja las operaciones primitivas identificadas en el capítulo 2. Mientras, las regiones como sujetos atómicos recogen las propiedades que determinan su configuración y por ende su comportamiento.

3.5 Presentación de escenarios principales

Bajo este apartado se procede a la exposición y desarrollo de aquellos escenarios considerados de interés en la elaboración del sistema y cuya elección pretende mostrar el flujo que garantiza el cumplimiento de los objetivos marcados por los participantes. En estos no se revela información acerca de la representación interna de los componentes del sistema. La tarea se apoya en los llamados diagramas de secuencia, que

permiten capturar la relación entre distintos participantes –componentes, actores o subsistemas entre otros– tanto del problema como de la solución ofrecida para un escenario concreto. Se evidencian así los mensajes y señales intercambiadas entre ellos y permite visualizar el tiempo de vida de un participante, cuál de ellos inicia cierta acción, cómo interactúan entre sí y quién hace qué, entre otros aspectos. Asimismo es posible refinar y depurar este tipo de diagramas en sucesivas iteraciones o fases del desarrollo, de modo que se amplia la información contenida, cerrando la brecha existente entre el problema y la solución. Si se requiere profundizar entre todas las posibles ejecuciones para un mismo caso de uso, entonces sería necesario acudir a los denominados diagramas de máquinas de estado –también permiten representar acciones generadas dentro del sistema y los cambios inducidos– en lugar de los aquí empleados. La aproximación empleada para su representación no obedece a un control centralizado o distribuido, hecho que resulta habitual para la expresión de los flujos, sino que se inspira en diagramas existentes en [56] y [57], donde una línea de vida constituye el sistema, que interacciona con los distintos actores, considerados en este caso actuadores por lo que podrán recibir señales y mensajes, así como enviarlas.

La primera de las figuras 3.4 refleja cómo una tarea de la dimensión de usuario se sirve, mediante una solicitud, de los servicios del sistema para completar su actividad y en concreto para el aspecto aquí representado, el escenario en el que una tarea cede su tiempo de ejecución si su acción se da por concluida. El participante que refiere al mecanismo de escalado de privilegios se considera la capacidad ofrecida por la plataforma, cuyos detalles *a priori* no son conocidos, para el correspondiente cambio de dimensión.

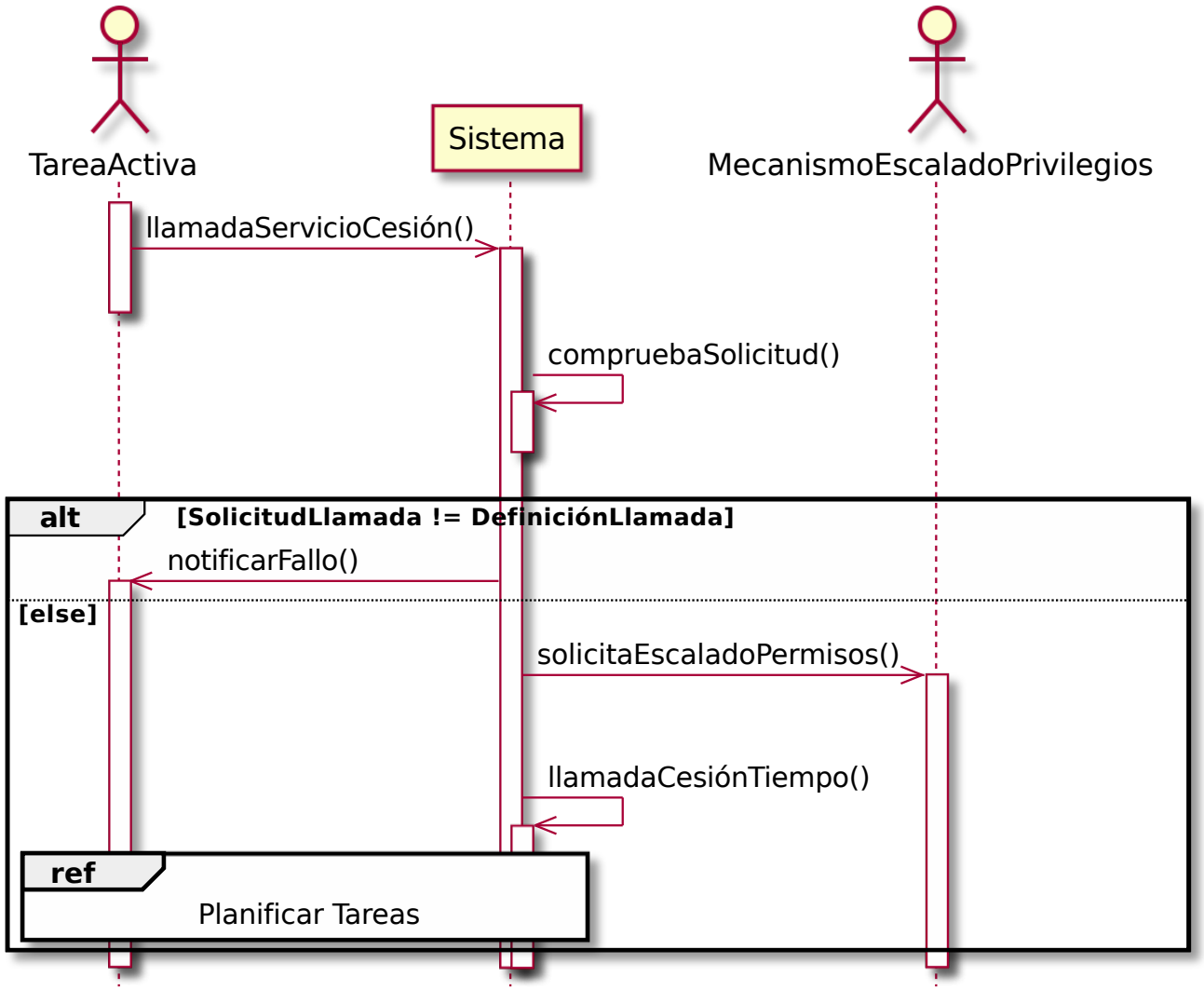

FIGURA 3.4: Escenario que recoge la llamada de sistema para cesión de tiempo.

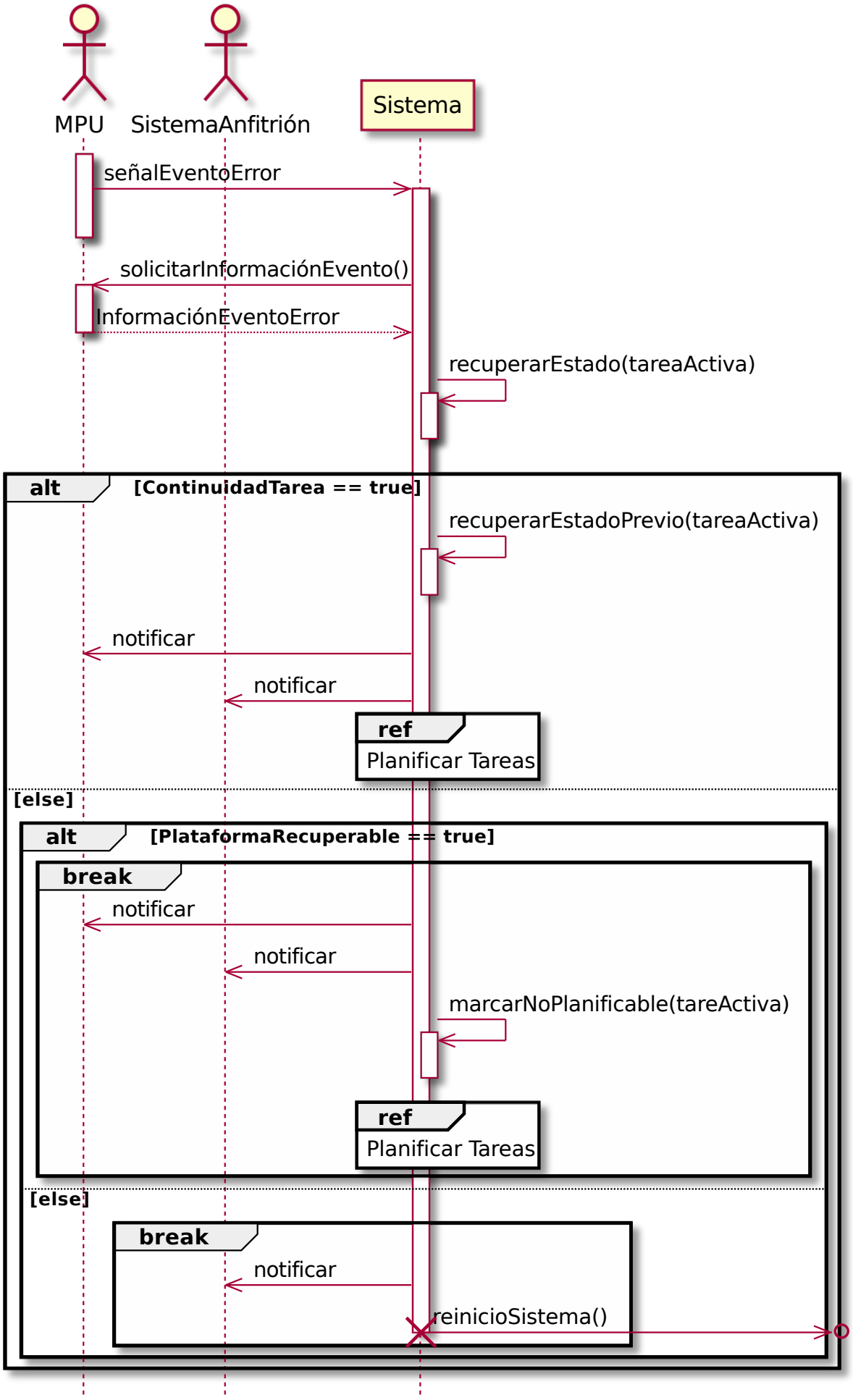

FIGURA 3.5: Escenario que describe el flujo previsto en condición de fallo.

La figura 3.5 representa el flujo seguido en condición de fallo de acceso que tiene como destino una posición fuera del dominio de protección de la tarea activa. En este caso resulta de interés conocer el origen y la causa de fallo, con objeto de determinar el alcance del mismo una vez hace su aparición. Si puede garantizarse la continuidad de la tarea, entonces se procede a restaurar su estado previo al error. Asimismo, si el cambio inducido en el sistema resulta recuperable entonces, se planifica una nueva tarea, mientras la anterior se marca como no planificable para futuras iteraciones. En caso contrario será necesario recuperar el estado inicial y viable para la ejecución del sistema, hecho que requiere de un reinicio completo. La notificación al sistema anfitrión en existencia de fallo es de obligado cumplimiento.

Tal y como describe el caso de uso, la planificación de tareas presenta tres puntos de entrada, hecho diferenciable por el actuador que comienzo a este. En el escenario 3.6 se presenta la opción por agotamiento de cuanto. Tal y como se observa, el flujo da comienzo tras la interacción del actor temporizador, esto es, tras recibir la señal del evento programado. Tras ello, el sistema conserva el contexto de la tarea activa, a la que se le retiran los recursos asignados para la ejecución. El flujo se servira de la funcionalidad prestada por los escenarios restantes presentados en este apartado. Previo paso a la delegación de control, el sistema deberá asegurar que el temporizador se encuentra activo para la siguiente ejecución.

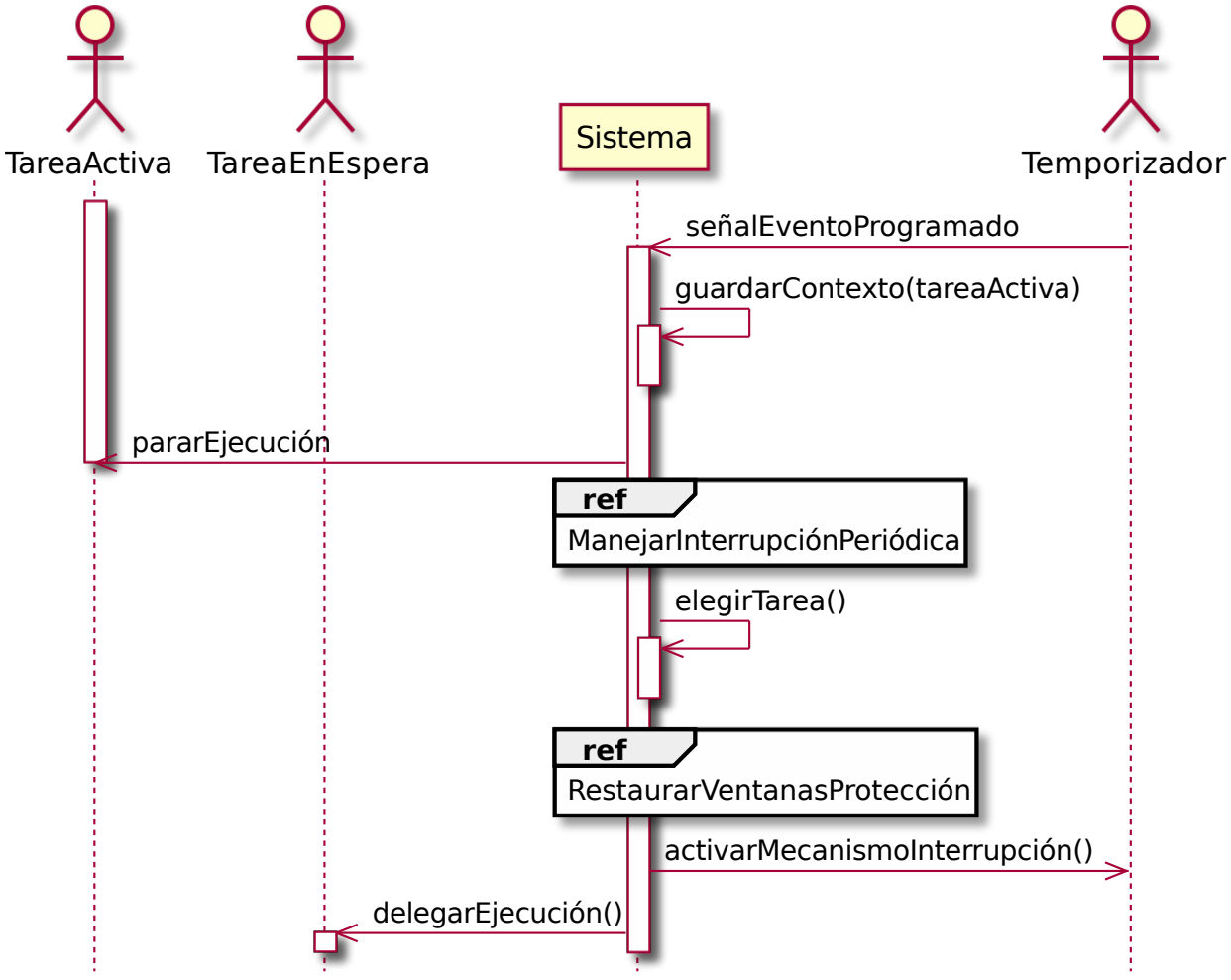

FIGURA 3.6: Escenario que recoge la planificación de tareas tras agotamiento de cuanto.

La figura 3.7 describe el escenario para la configuración del temporizador ante la planificación por agotamiento de cuanto. No obstante, este flujo se realiza con independencia del punto de entrada producido. Este escenario presenta cierta genericidad, en tanto en cuanto la actividad es la misma con independencia de la interrupción a manejar. Así pues, ante un incremento de las interrupciones soportadas sólo será necesario comunicar el valor en unidades de tiempo asociado al tipo de interrupción, manteniendo el flujo aquí provisto.

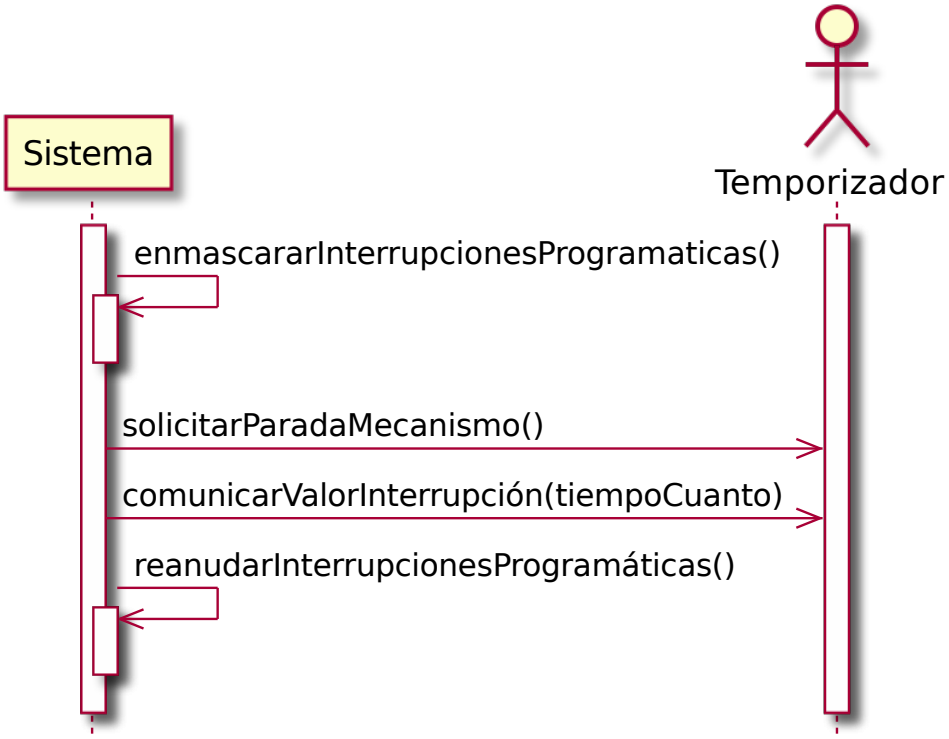

FIGURA 3.7: Escenario que refleja la interacción con el temporizador para la programación de tiempo de cuanto.

Por último, la figura 3.8 representa la acción de intercambio de dominio durante la planificación, de forma que el resultado es el punto de vista de protección para la tarea escogida como candidata. Para ello, se consulta la información disponible relativa a las regiones de dicha tarea hasta agotar todas y cada una de las entradas. La asignación de cada región de protección se considera una operación fundamental, esto quiere decir, que si alguna de ellas falla, la ejecución no puede continuar y ante la negativa será necesario proceder a su corrección, tal y como expresa el flujo alternativo previsto para el correspondiente caso de uso.

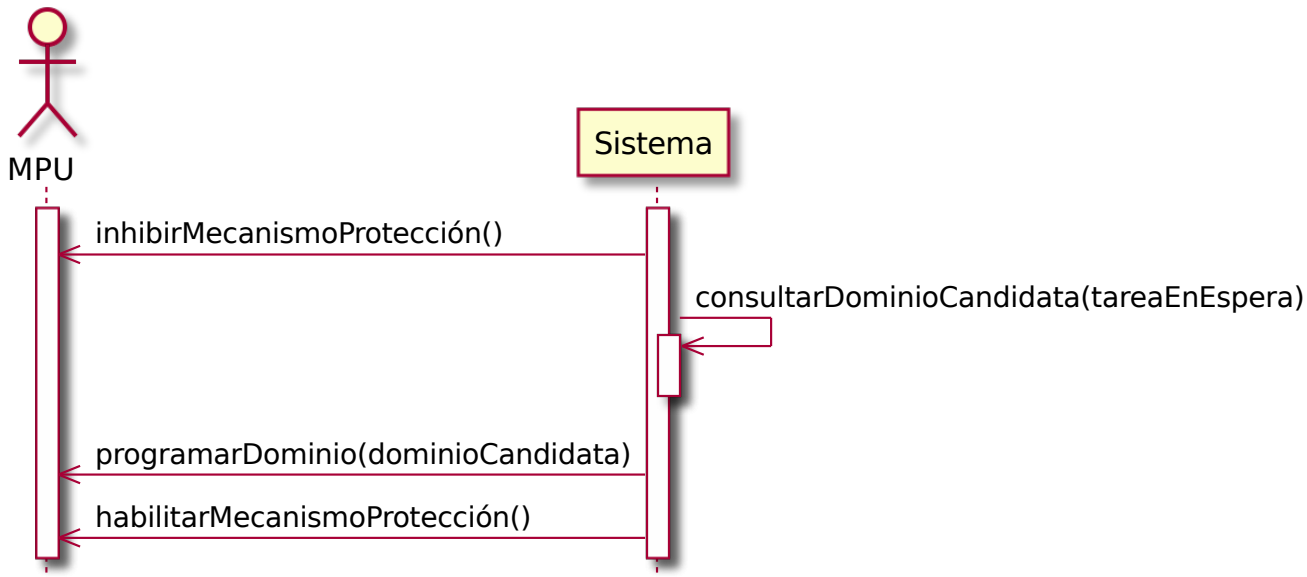

FIGURA 3.8: Escenario que expresa la interacción para el intercambio de dominio de protección.

3.6 FLUJO PRINCIPAL DEL SISTEMA

Tras la exposición de la descripción genérica, como narrativa del contexto de este trabajo, y los requisitos que determinan tanto qué debe hacer el sistema como los puntos de interacción con las entidades externas, surge la necesidad de estudiar el comportamiento relativo aquellos procesos propios de la solución. Conviene aclarar la existencia de una casuística particular para este proyecto, en el que se referencian conceptos propios del dominio del problema con terminología generalmente asociada a la implementación y por ende a los sistemas de información. Este hecho, que puede inducir a la confusión, puede revertirse mediante el uso de de los denominados diagramas de actividad, en tanto en cuanto resultan la opción optima, dada su capacidad para agrupar flujos de actividades ordenadas para cualquier nivel de abstracción o punto del desarrollo, para la representación de una primera aproximación al sistema.

La figura 3.9 comienza mediante una posible representación del estado confiable y viable para la ejecución de tareas –previo al propio sistema–, hasta alcanzar el ínstante en el que tienen lugar las acciones desencadenadas por las entidades que hacen uso del sistema.

La especificación *UML* establece que cada acción proporciona a su salida información, en forma de *token*, cuya llegada, supeditada a la evaluación de una condición, da comienzo a las sucesivas acciones.

Así pues, existe un flujo principal cerrado en el que se planifican tareas, proporcionando así acceso a los recursos de la plataforma, de modo que estas puedan completar sus fines, y asignando el correspondiente dominio de protección. Si no existen tareas que requieran de ningún servicio, entonces se desemboca en un estado de reposo, reflejado en este caso como una tarea de sistema. De manera concurrente a la planificación se configurará el temporizador que determinará el final de la ejecución para la tarea –si esta no cede voluntariamente el control de la plataforma–. Asimismo se contempla un ciclo, cuya repetición tiene lugar tantas veces sea requerido durante el tiempo de vida de la tarea, tal que refleja la solicitud de servicios de sistema bien mediante una llamada explícita o mediante mecanismos particulares de las arquitecturas y que no no resultan de relevancia en este apartado. Se plantea también un flujo alternativo, fruto de una condición anormal en la ejecución, cuyo resultado provoca un desmembramiento en tres ramas, donde la primera entronca de nuevo con el flujo principal, la segunda fuerza el reinicio de la plataforma –por tanto inicia de nuevo el flujo– y la tercera finalizaría la actividad global sin posibilidad de continuación. Por último se concibe la posibilidad de establecer una región de fondo por defecto durante el arranque, tal que esta ventana cubriría rangos de memoria considerados como críticos en ausencia de los dominios asignados para cada tarea. Si bien es cierto, este aspecto se enmarca como precondición para el propio sistema, pero se considera su incorporación en el diagrama, puesto que refleja otro uso efectivo de la unidad de protección.

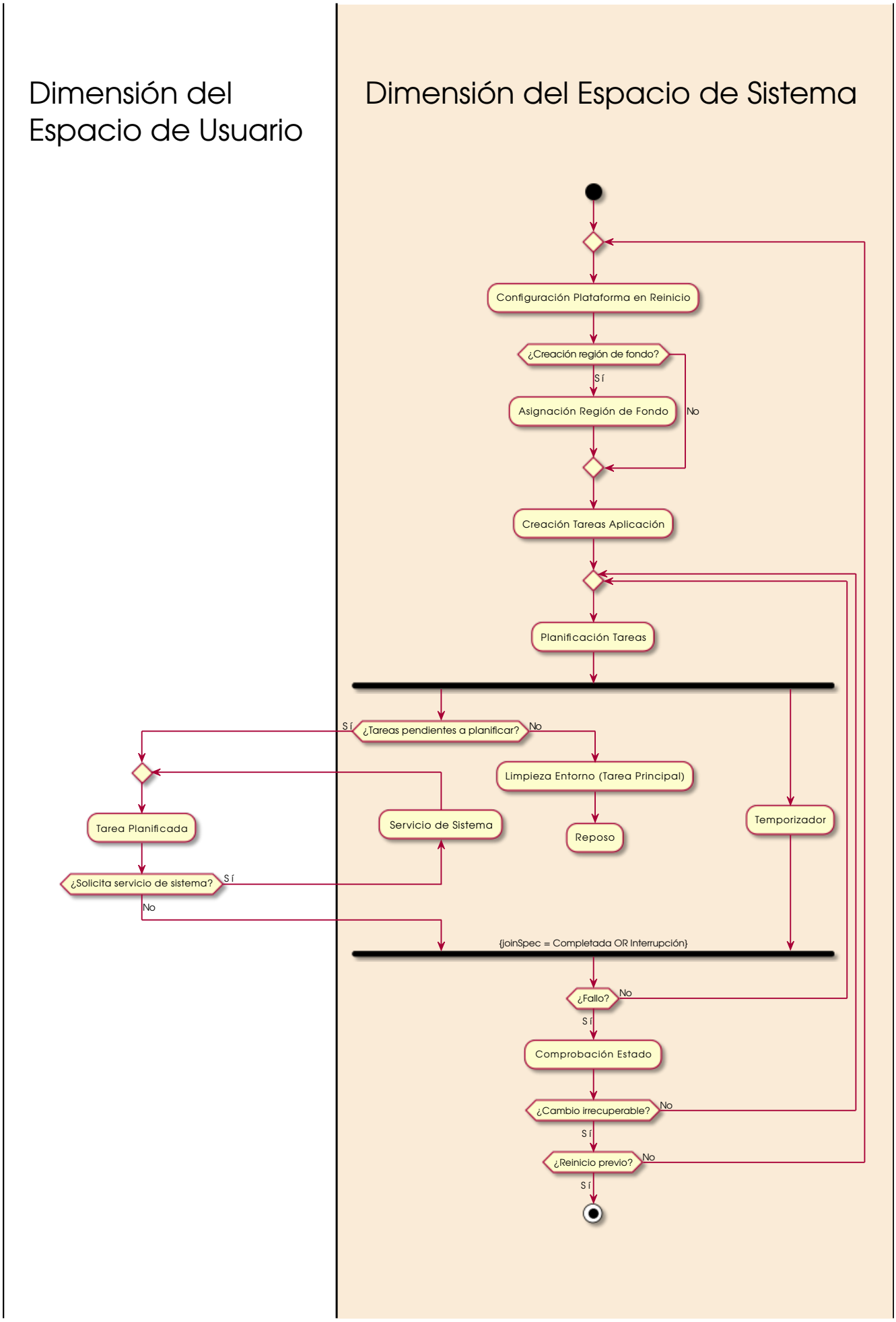

FIGURA 3.9: Flujo general del sistema

3.7 HEURÍSTICA DE PLANIFICACIÓN DE TAREAS

Se presenta en este apartado una posible solución para la gestión de procesos concurrentes:

Data: tablaTareas ← Estructura que contiene la información estática de las tareas
Data: tareaActual, tareaPlanificada ← Ambas representan una entrada de la estructura tablaTareas, por lo que resultan distintos puntos de vista en los que puede encontrarse una tarea
Result: La acción de la heurística genera el punto de entrada, a partir del cual continua el flujo

```
/* Se procede a la conservación del contexto de la tarea activa -entre los
   registros se incluye el contador de programa o PC-                    */
for i ← 0 to numRegistrosProcesador do
    tareaActual.Registros[i] ← iRegistroProcesador
end
tareaActual.EstadoProcesador ← registroEstadoProcesador
tareaActual.EstadoTarea ← Inactiva
baremoPrioridad ← 0
/* Cada entrada de la estructura representa una tarea                    */
for i ← maxNumTareas to 0 do
    if tablaTareas[i].EstadoTarea != noDisponible then
        tablaTareas[i].Ansiedad ← tablaTareas[i].Ansiedad + tablaTareas[i].Prioridad
        if tablaTareas[i].Ansiedad > baremoPrioridad then
            baremoPrioridad ← tablaTareas[i].Ansiedad
            tareaPlanificada ← tablaTareas[i]
        end
    end
end
tablaTareas[i].Ansiedad ← 0
tareaPlanificada.EstadoTarea ← enEjecución
registroEstadoProcesador ← tareaPlanificada.EstadoProcesador
/* Se restaura el contexto de la nueva tarea, estableciendo así el nuevo punto
   de entrada para la ejecución                                          */
for i ← 0 to numRegistrosProcesador do
    iRegistroProcesador ← tareaPlanificada.Registros[i]
end
return
```

Heurística correspondiente a la planificación de tareas.

CAPÍTULO 4

DISEÑO DEL SISTEMA

Bajo este capítulo se proponen aquellos diagramas en representación de la lógica a desarrollar en este trabajo. En concreto, se reflejan, dados unos criterios, distintos puntos de vista considerados para el *software*, de modo que por un lado se presenta la disposición del sistema en virtud de la abstracción respecto de los elementos físicos de la plataforma, mientras que por otro se muestra en detalle las entidades que lo conforman, incluyendo las operaciones y propiedades para cada una, así como las relaciones existentes entre estas. Cada uno recibe su propio apartado, en donde se discutirán algunas de las aproximaciones tomadas para el diseño presentado.

4.1 ARQUITECTURA LÓGICA

La arquitectura lógica permite la estructuración de los constructos *software* con independencia de su disposición real. Así pues, varias entidades, a pesar de residir en distintos sistemas físicos, pueden constituirse en un único servicio, hecho que además resulta transparente para los clientes que interactúan con este, en tanto en cuanto desconocen la ubicación de cada elemento. El patrón arquitectónico de cinco capas, descrito en [58], permite la agrupación de aquellos componentes que comparten un conjunto de propiedades y funcionalidad similares. No obstante, su principal atractivo es la capacidad para distribuir cada elemento con respecto a la noción que tienen sobre la plataforma física. De esta forma, cada capa se agrega sobre la anterior, la cual pone a su disposición sus servicios, presentando por un lado una modularización del sistema, así como una mayor abstracción a medida que se asciende de nivel. A continuación se enumeran sumariamente cada una de las capas proyectadas por este patrón:

- Capa de aplicación, que recoge aquellos elementos que conforman las tareas de usuario. Se encuentran en el nivel más elevado de la abstracción, por lo que requieren del resto de dominios para completar su tarea.
- Capa de interfaz, tal que ofrece aquellos componentes encargados de presentar la interfaz gráfica para la mediación con el usuario.
- Capa de comunicación, la cual se encuentra separada de la capa de sistema operativo y actúa como *middleware* para la comunicación entre tareas y servicios, manejando canales de comunicación y diversos protocolos.
- Capa de sistema operativo, en donde se disponen aquellos servicios privilegiados y que afectan a la distribución y control de recursos. Interactúa con la capa de abstracción *hardware* para llevar a cabo su función.
- Capa de abstracción *hardware*, en ocasiones conocida como *BSP* (*Board Support Package*) y comúnmente provista por el fabricante. Esta se dispone en el nivel más bajo de abstracción, por lo que conoce

todos los detalles relativos a la plataforma física. Generalmente alberga aquellos componentes asociados a la configuración tanto de periféricos como de la unidad de procesamiento, las rutinas de gestión de eventos y además contiene la rutina de control para el reinicio (*bootloader*).

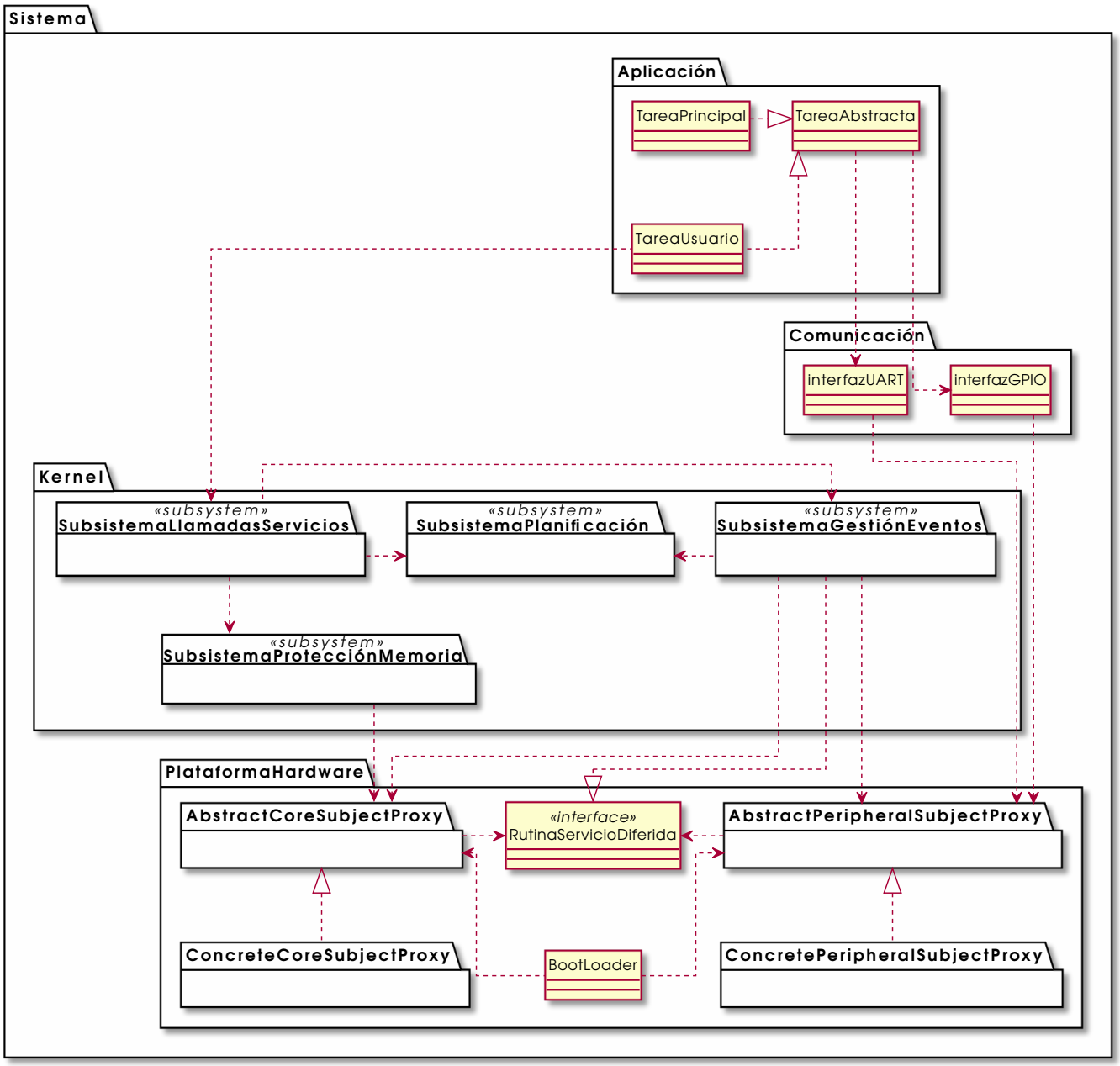

FIGURA 4.1: Diagrama de paquetes como representación de la arquitectura del sistema

Tal y como muestra la figura 4.1, el diagrama de paquetes [59] resulta la opción elegida para la realización de la arquitectura, decisión respaldada por su capacidad para la representación de dominios y relaciones entre los distintos componentes que los conforman. Asimismo, cada componente se asocia junto a otros en subsistemas, tal que estos refleja una organización del *software* en función de características comunes a nivel particular, de modo que para cada dominio se distinguen objetos con la mayor cohesión entre sí. Cada subsistema es estanco, por lo que que tan sólo proporcionará interfaces para su interacción, recibiendo peticiones, las cuales son delegadas a sus partes internas, a través de ellas y devueltas por el mismo punto. Cabe destacar que en este punto no se precisa necesaria la información de despliegue para cada uno de los elementos en el dispositivo (mediante un diagrama de componentes), en tanto en cuanto la plataforma escogida sólo dispone de una unidad de procesamiento, así como un espacio de direcciones físicas –común para todos los componentes del sistema–. Se atisban dos tipos de relaciones para este diagrama [60], donde

la primera expresa dependencia –representada por una línea discontinua con una flecha cerrada– y se interpreta como una relación semántica entre elementos del modelo, donde la modificación en las características del proveedor requieren un cambio de significado para el cliente. Mientras, la segunda se define como realización –generalmente recae en un mismo dominio y se representa como una línea discontinua con flecha abierta– e indica asociación entre un elemento que establece una especificación y aquel que implementa su funcionalidad.

En este caso, la denominada capa de interfaz se encuentra ausente, ya que a efectos prácticos, la interacción con el usuario se limita al uso de puertos serie e interfaces de entrada y salida genéricas, cuya responsabilidad de gestión recae en la capa de comunicación. Su labor consiste en atender las peticiones recibidas con intención de ser transmitidas al correspondiente receptor (otro sistema o tarea), creando y manteniendo para ello los correspondientes enlaces y mediando entre los formatos que entienden la aplicaciones, así como aquellos establecidos por los protocolos empleados. La separación de tal responsabilidad respecto del *kernel* obedece a la motivación de reducir su extensión y por ende la complejidad en su desarrollo y mantenimiento. Se podrían así incluir implementaciones provistas por terceros y compatibles con estándares bien definidos, de modo que las aplicaciones seguirían formatos conocidos para la comunicación. En contraste, tendría lugar una solución *ad hoc*, que obligaría en este caso a la inserción de un adaptador entre las tareas y el servicio –aunque esta aproximación podría resultar de utilidad al ampliar la funcionalidad provista, por ejemplo, aportando un sistema de redundancia previa a la transmisión del mensaje–. Para esta segunda propuesta, su realización podría orientarse dinámicamente, esto es, una tarea planificable, o en su defecto estáticamente, tal que se elegirían sólo aquellos elementos requeridos por las naturaleza del problema, logrando así un menor tamaño de binario generado. Sin embargo, la separación de servicios para la comunicación no se encontraría exenta de inconvenientes. En concreto, la existencia de acoplamiento con el sistema operativo seguiría presente para aquellos casos en los que los periféricos demandan un estado de ejecución privilegiada para su labor, así como en situaciones para las que se desconoce bien el destinatario del mensaje o el dispositivo físico encargado de la transmisión. En cualquiera de estas circunstancias se exigirá la presencia de una entidad que interceda entre las correspondientes capas involucradas. Este hecho es despreciable para el trabajo aquí propuesto, puesto que las opciones para la comunicación presentes en la plataforma son conocidas a priori y no requieren de permisos especiales.

Llegado el momento, los componentes correspondientes al resto de capas se relacionan, de una manera u otra, bien directa o indirectamente con las entidades *hardware* que constituyen la plataforma. Dichas entidades no son visibles bajo este diagrama, sino que en su lugar se reflejan como abstracciones lógicas, las cuales actúan como puntos de acceso y encapsulan la implementación que permite su correcta configuración y manipulación. Para ello se empleará el denominado patrón *Hardware Proxy* [35] –descrito con mayor detalles en el siguiente apartado–, que logra mitigar el impacto que tiene sobre los clientes cualquier modificación del dispositivo *hardware*. Es posible establecer diversas clasificaciones de *proxies* en categorías atendiendo a distintas características de los componentes que constituyen la plataforma física –por ejemplo si se registran o no en el espacio de direcciones físicas–. Aquella escogida para este desarrollo atenderá a las necesidades de escalado de privilegio para su correcto funcionamiento. Como indica el diagrama, tanto la capa de comunicación como aquella referida al *kernel* hacen uso de los dispositivos a través de las interfaces que disponen para tal fin. El comportamiento interno de cada *proxy* vendrá dado por su implementación, a la que generalmente se conoce como *driver* de dispositivo, expresada como una relación de realización sobre dichas interfaces. Entre la funcionalidad que incluyen se encuentran aquellas operaciones para la conversión de datos a la codificación comprendida por el dispositivo, los gestores de interrupción. En lo que atañe a la propuesta de este trabajo pueden encontrarse capacidades particulares de las arquitecturas estudiadas,

esto es, todas aquellas referidas a la interacción con la unidad de protección de memoria o al mecanismo de escalado de privilegios. Asimismo, durante el arranque de la plataforma física todos sus elementos deberán ser coordinados en un orden preestablecido por el fabricante, con objeto de proporcionar un estado confiable para la ejecución del *kernel* y en consecuencia de la aplicación, donde dicha tarea recae en el denominado *bootloader* o gestor de arranque –comúnmente provisto por el fabricante junto al resto del *BSP*–. Las responsabilidades asignadas a este *bootloader* dependerán de las circunstancias concretas del sistema a desarrollar. Así pues, si el sistema operativo soporta una instalación de dispositivos dinámica, el registro inicial de los dispositivos existentes junto a sus correspondientes *drivers* será obligación del mencionado gestor de arranque. Esta aproximación aumentaría el acoplamiento entre ambas capas, generando además dependencias tanto en sentido descendente y como en sentido ascendente. Este efecto puede mitigarse mediante el uso de un *Mapper*, que oculta la existencia de cada uno de los componentes, pero que a su vez mantiene la apariencia de comunicación directa. No obstante, dado el carácter estático que se pretende conseguir para este trabajo, dicha instalación es delegada al propio proceso de construcción *software* –en concreto durante la compilación–. Se perdería la capacidad para acoplar o sustituir periféricos bajo demanda, pero se logra a cambio una menor latencia tanto en el arranque como en la gestión de servicios que incurren en el uso de los dispositivos que ofrece la plataforma.

Por defecto, cualquier *driver* de dispositivo se ejecutará, en primera instancia, bajo el mismo modo que aquel en el que lo hace la aplicación, la cual indirectamente requiere de sus capacidades. Sin embargo, tal y como se ha señalado, en ocasiones resulta inviable hacer uso de los recursos en ausencia de privilegios, por ejemplo en acceso a objetos compartidos –controlados por *integer object handlers*–. Con este motivo un *kernel* puede poner a disposición los denominados servicios de sistema, en los que delegan las tareas y cuya función permite el control de los recursos de la plataforma –aunque si bien es cierto, algunos sistemas como *Linux* [61] ofrecen a las aplicaciones las denominadas *capabilities*, sin necesidad de mediación alguna–. Dichos servicios no realizarán el correspondiente escalado de privilegio, sino que en primer lugar comprobarán los permisos que el origen tiene sobre un recurso, haciendo uso por ejemplo de una estructura *ACL –access control list–*, tal y como recoge [62], propuesta que además regula el acceso a la rutina que valida tales permisos mediante la unidad de protección –hecho que sugiere una aproximación para la invocación de servicios de sistema, que será comentada en el capítulo referido a líneas de trabajo futuras–. Del mismo modo, la lista de permisos podría ser protegida por la propia *MPU*, indicando su lectura pero no así su modificación, puesto que aún se permanece en el modo de ejecución de la tarea. Ante la negativa, tal y como se dictamina durante la concepción de la solución, se procederá a la notificación de fallo, hecho representado en la figura 4.1 como relación de dependencia con el subsistema gestor de eventos. Mientras, en caso afirmativo se procedería al cambio de modo, el cual se encuentra estrechamente ligado a las capacidades de la arquitectura. Puede deducirse por tanto que el subsistema ofrecido para la invocación de servicios no atenderá por sí solo las peticiones, sino que delegará al mecanismo real, el cual generalmente se localiza en la capa *AbstractHW*. De esta forma, se logra desacoplar a la aplicación en invocación de servicio de detalles propios de las arquitecturas, puesto que simplemente deberán ajustarse a la interfaz que proporciona para ello el sistema operativo. Si bien es cierto, algunos lenguajes de programación ofrecen en sus librerías soluciones –conocidas como *Wrappers*– con este fin, pero generalmente estas dependen de implementaciones de terceros o no contienen todas aquellas definidas por el *kernel* o bajo estándares. Así pues, esta opción se adecuaría correctamente a incrementos de la funcionalidad con independencia de la plataforma o el lenguaje utilizado. Entre los servicios previamente identificados se encontrarían aquellos que habilitan la planificación de tareas en cesión de tiempo o a la asignación de protección para un objetos pertenecientes a tarea. Este efecto se plasma en el diagrama mediante como una relación de dependencia existente hacia los subsistemas de planificación de tarea y protección de memoria.

El subsistema *EventHandling* no sólo se destina al control de interrupciones, aunque si bien es cierto resulta su capacidad más relevante, sino que también abarcaría conceptos como eventos y operaciones programadas. Tal y como se indica, las rutinas de gestión en interrupción se recogen en el *BSP* -dada la estrecha relación existente con las definiciones propuestas por las arquitecturas, esto es, el orden de prioridad de cada tipo y su número total generalmente incompatibles entre sí. Por ejemplo, para la arquitectura elegida en este trabajo, el fabricante ofrece un módulo *hardware*, con objeto de disminuir la latencia en procesamiento de excepción, de modo que es capaz de enlazar cada una de las posibles excepciones, sin necesidad de intervención *software*, con su correspondiente rutina de servicio. Del mismo modo ocurre para la tabla de vectores, la cual generalmente vendrá provista por el propio *BSP*, aunque en ocasiones, como apunta [58], puede ser proporcionada por el propio sistema operativo –limitado a soluciones dinámicas de instalación de dispositivos que no obstante deberán conservar una tabla estática inicial en caso que sea necesario su restauración–. A efectos de este desarrollo, y del mismo modo que para la gestión de dispositivos, se optará por una resolución de rutinas de servicio en tiempo de compilación. Este subsistema actuaría de igual modo que el dedicado a la interceptación de llamadas a sistema, por lo tanto como un *Mapper*. Asimismo, resulta de vital importancia su conjugación con aquel destinado a la protección de memoria, con la finalidad de identificar la condición de fallo en acceso controlado por la *MPU*.

4.2 Representación de los componentes del sistema

Una vez delimitada la distribución para las entidades identificadas, en base a una funcionalidad común y su conocimiento respecto de la plataforma física, se procede al estudio en mayor profundidad de las relaciones en las que participan. En este caso, el diagrama de clases será aquel elegido con esta finalidad, esto es, la representación inicial del prototipo *software* a elaborar y sobre el cual se realizará la traducción al código final. Estos se caracterizan por su capacidad para reflejar relaciones existentes entre los elementos (cada uno como descripción de conceptos del dominio de la solución sobre los que se organiza el diseño) que conforman la solución, así como el número total de ellos involucrados en cada relación. Asimismo, consiguen describir para cada elemento tanto sus propiedades como sus operaciones y la visibilidad de estas. No obstante, es frecuente la aparición de este diagrama en desarrollos que incluyen el denominado paradigma de orientación a objetos [59], por lo que en este caso se hará uso de aquellas características genéricas y estereotipos (por ejemplo, <*File*> como representación de fichero fuente o <*PDS*> como estructura de datos pasiva, esto es, aquellas que no contienen operaciones) con la finalidad de eliminar dicha connotación, en tanto en cuanto no es este el paradigma elegido para este trabajo. La relación más extendida entre los elementos se conoce como asociación y se define como el enlace de comunicación disponible entre las entidades, mientras que el resto de relaciones presentan una función meramente descriptora de características.

Como puede observarse, la figura 4.2 se corresponde con el resultado provisto como solución a este proyecto y cuya composición viene determinada, en parte, por la incorporación de patrones de diseño. Tales patrones ayudan a resolver problemas específicos, identificados en el contexto de los sistemas empotrados, mediante aproximaciones genéricas pero efectivas en su tarea. Así pues y en lo sucesivo se describirán algunos de los patrones empleados.

Tal y como se describe durante el análisis, resulta necesario el uso de algún tipo de estructura que permita identificar cada una de las tareas provistas para la aplicación, así como aquellos elementos que hacen referencia a su estado, de forma que pueda retomarse la ejecución si así se desea. Es por ello que este diseño se beneficia en gran medida del patrón de prioridad estática [58], el cual mapea en el sistema toda la información necesaria de un proceso –entre la que se incluye la pila– con objeto de su planificación, he-

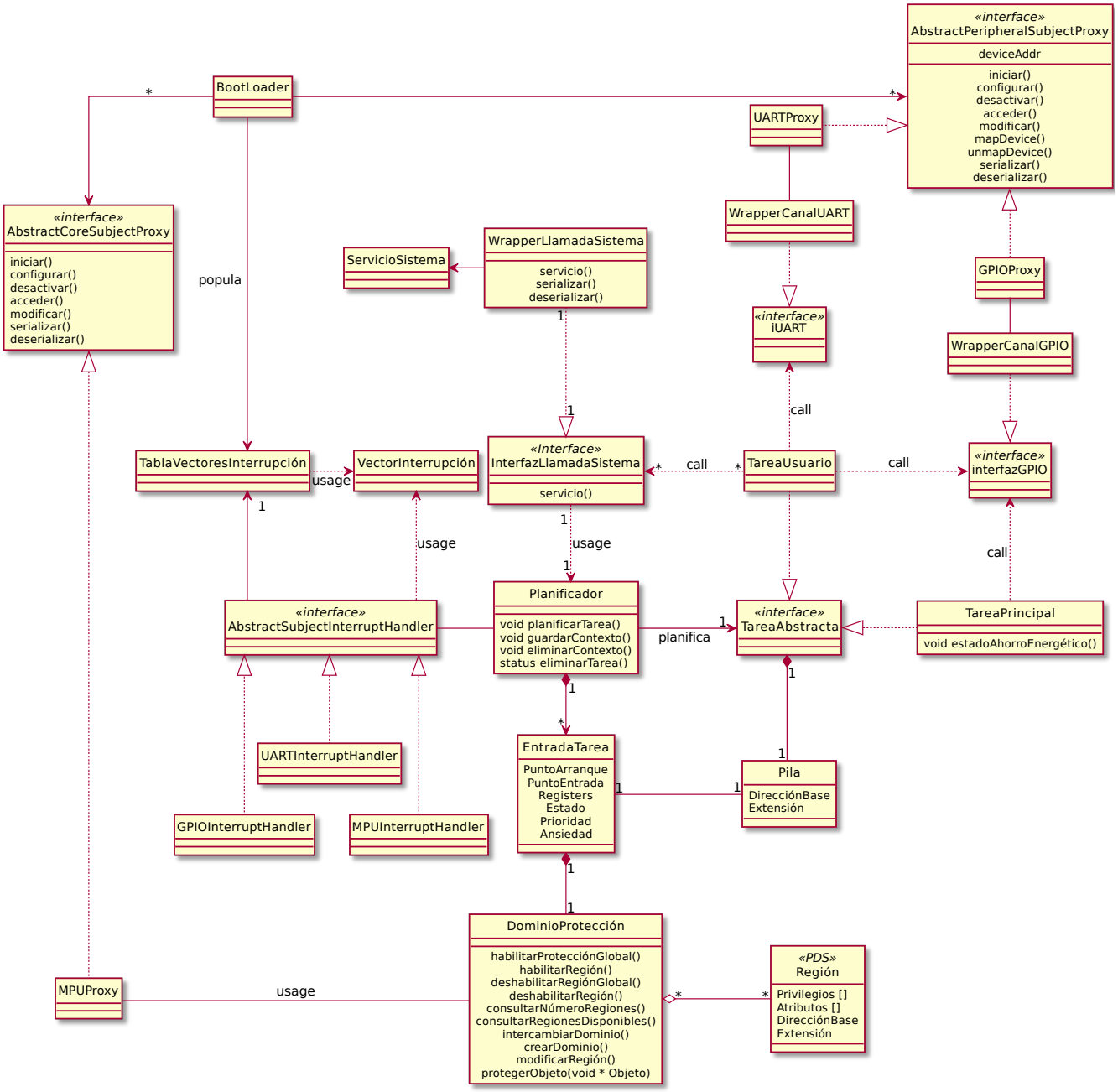

FIGURA 4.2: Diagrama de clases en representación del diseño del sistema

cho que queda reflejado por las entradas que constituyen la estructura del bloque de control de tareas. Cabe mencionar que el patrón se ajusta a las necesidades de este problema, por lo que se altera en consecuencia, incorporando el correspondiente dominio de protección, el cual además habilita la mediación ante peticiones asociadas a la protección de memoria (siendo las posibles acciones aquellas previamente definidas como operaciones primitivas). La configuración de tales dominios resulta de la agregación de las regiones (como elementos atómicos), las cuales dadas la naturaleza de este prototipo son conocidas *a priori*, por lo que su representación se llevaría a cabo mediante estructuras no mutables, pero deberán permitir su lectura ante un cambio de contexto.

Puesto que la interacción con la unidad de protección es propia para cada arquitectura, el dominio delegará a los correspondientes componentes dicha acción. Así pues, este actuará como un *Wrapper* [35], transformando si fuese necesario –mediante operaciones internas no visibles de *marshalling/unmarshalling*– la información disponible al formato adecuado. El comportamiento para la invocación de llamadas se asemeja al anterior descrito, por lo que de nuevo involucra la actuación de un patrón adaptador. *SystemCallWrapper* recibirá aquellas peticiones generadas por las tareas en apariencia del servicio real, pero delegará a la correspondiente implementación de la llamada prevista (proporcionada por un tercero o el propio desarrollador). En la misma medida, las tareas no interactúan de forma directa con los periféricos destinados a la comunicación con el entorno. Requieren por tanto de los *Wrappers* de comunicación (*UARTChannelWrapper* y *GPIOChannelWrapper*), que a su vez delegan a las correspondientes rutinas de dispositivo, responsabilizándose de la conversión de información e incluso aportando mecanismos para la redundancia.

La interacción directa con los componentes *hardware* estaría mediada por sus abstracciones, siendo estas aquellas entidades establecidas bajo el patrón *Hardware Proxy* [35]. La inclusión de dichos elementos al modelo no añade valor si la funcionalidad provista por el *BSP* satisface las necesidades del problema, en tanto en cuanto su realización puede ser ignorada –el sistema únicamente deberá adecuarse a las interfaces que estos proveen–. No sucede de esta forma para este prototipo para el que ha sido necesaria la alteración de ciertas secciones correspondientes a la lógica del fabricante y en el peor de los casos la creación de elementos *ad hoc* –tal y como se describe en el apartado de adversidades para el capítulo 5–. Este tipo de patrón establece una serie de operaciones genéricas válidas para la manipulación y configuración asociada a cualquier componente *hardware*. Es posible la representación dinámica, mediante una estructura de datos, del estado de cada dispositivo para un momento de la ejecución, cuyo empleo resulta de utilidad si se requiere su presencia bajo demanda. No obstante, esta característica quedaría sin efecto para este proyecto, dado el carácter estático. Aun así, la existencia de un atributo que indica la posición concreta de instalación en el espacio de direcciones de la plataforma resulta de utilidad en este trabajo, con la finalidad de determinar la posición fija de los componentes provistos por el fabricante. Junto a las entidades contempladas por el patrón se añade la existencia de los gestores concretos de interrupción y dependientes de cada plataforma. Además, dado el esquema utilizado, el cual distingue entre componentes con y sin privilegio en la interacción, se pergeñe una operación que comprueba el modo de ejecución cuando así sea necesario. Por último, cabría resaltar la existencia de una rutina de inicio, basada en el patrón *Mediator* [35], capaz de coordinar los distintos componentes en el orden establecido por el fabricante. Se consigue de este modo reducir la complejidad de sincronización entre componentes, al delegar la responsabilidad bajo un único punto.

CAPÍTULO 5

EVALUACIÓN MEDIANTE PRUEBA DE CONCEPTO

Tras la exposición de los contenidos correspondientes a las actividades previas, el siguiente paso corresponde a la implementación del prototipo que permitirá evaluar las bondades del mecanismo de protección estudiado. A lo largo de esta fase se presentan dos planteamientos que constituyen la simulación de ejecución para una aplicación y que facilitan la extracción de información relativa a las características de uso de la *MPU*. El primero de ellos se centrará en el efecto que la acción de desbordamiento de pila de usuario tiene sobre la estabilidad de la plataforma y como el uso de subregiones permitirá evitar su aparición. Mientras, el segundo se limitará a la prevención de condiciones anormales de flujo en el ámbito de la ejecución especulativa de código, siendo esta circunstancia propia de la microarquitectura escogida y con en su entorno de aplicación.

Puesto que la plataforma soporta la ejecución de código desde la memoria persistente se considerará a la propuesta como un sistema *XIP* (*Execute In Place*). Dado que no se requiere la copia previa de este contenido a la memoria principal, esta se destina únicamente al conjunto de objetos dinámicos tanto de las tareas como del *kernel*. Así pues, con motivo de facilitar la exposición del contenido provisto en este capítulo se caracterizará mediante un diagrama la distribución de dicho contenido en el espacio de memoria, así como la regiones de protección disponibles, cuyo esquema de colores se asocia a los puntos de vista relativos al *kernel* y a las tareas de usuario respectivamente.

5.1 DESBORDAMIENTO DE PILA

Generalmente, una tarea se compone como la agregación de varias rutinas establecidas en torno a una división modular de la lógica, con el fin de mantener cierta independencia entre los componentes. El flujo de la lógica determina la operación a realizar en cada momento, hecho que a veces requiere la invocación del módulo adecuado que la contiene. Dicha invocación se rige mediante una estándar definido por el proveedor de la arquitectura, que indica, entre otros aspectos, la localización de los argumentos que recibe la rutina solicitada y el procedimiento para la construcción de la petición. Bajo el estándar definido para *ARM®–Procedure Call Standard for the Arm Architecture* o *AAPCS* [63]– se plantean dos posibilidades relativas al paso de argumentos. Por un lado, se requiere el uso de registros de la arquitectura específicamente indicados para este fin, mientras que si se excede este número, entonces se emplea la pila (*stack*) de la tarea. Si el comportamiento de la tarea resulta conocido de antemano en tiempo de compilación –el número de funciones que invoca y para cada una el número de parámetros requeridos–, es posible determinar de manera aproximada el tamaño requerido para la representación de la pila, por ejemplo, mediante pruebas que tienen como fin simular la ejecución de la aplicación. No obstante, es posible la existencia de condiciones no probadas o casos límite, frecuentes a medida que aumenta la complejidad de la lógica, que pueden poner a prueba la extensión de dicho espacio y en riesgo la estabilidad de la ejecución.

Una rutina se considera reentrante cuando su ejecución puede ser interrumpida en cualquier punto, garantizando la conservación de su estado antes de retomar el nuevo flujo. Por ejemplo, si la invocación a una rutina es recursiva, esta se invoca en sucesivamente sin completar su actividad hasta alcanzar el caso base definido. Cada llamada implica la conservación de estado, lo que demanda el uso de la pila, bajo la cual se inserta el estado concreto previo a la interrupción. Si además la rutina requiere del paso de parámetros mediante la pila, entonces su contenido también es volcado en el espacio. Con objeto de proseguir y dadas unas condiciones en las que la profundidad, esto es el número de invocaciones hasta encontrar dicho caso base, agota dicho espacio, entonces resultará necesaria la liberación de recursos. Cualquier invocación posterior provoca un efecto que se conoce como desbordamiento de pila, donde se emplea una dirección del espacio no contemplada para dicha tarea. La severidad e implicación de esta acción dependerá del ámbito de aplicación y los cambios inducidos. Es posible sobrescribir el contenido perteneciente a otra tarea o en la peor de los casos al sistema, con resultados recuperables para el primero y catastróficos para este segundo. Una tarea de la dimensión de usuario es recuperable, si y solo si se soporta su reinicio, esto es, es posible establecer de nuevo su configuración inicial para su posterior planificación. Con respecto al sistema, generalmente se concibe como irrecuperable, en la medida en que se altera su estado de manera global y sólo un reinicio de la plataforma permitiría solventar la situación, convergiendo a un estado conocido nuevamente. Si bien es cierto, existen lenguajes de programación que habilitan el control de este tipo de fenómenos, pero de entre estos, son pocos aquellos estandarizados por entidades de terceros y que además cumplen con requisitos y normas internacionales propias del ámbito de los sistemas embebidos. Por este motivo, resulta de interés la concepción de una técnica de protección en desbordamiento de pila, de manera que se garantice la continuidad en la ejecución, con independencia del lenguaje de programación empleado. Dados estos motivos, la unidad de protección de memoria se convierte en el mecanismo adecuado para la inmutabilidad del contenido en las posiciones accedidas, ya que por un lado permite la anticipación ante situaciones consideradas como irreversibles y por otro consigue establecer rangos de protección que delimitan el espacio de memoria útil para la pila, de manera que para cualquier operación que exceda el rango habilitado se generará una excepción.

Un vez establecida la concepción de la propuesta, se plantean a continuación dos procedimientos posibles para la protección en desbordamiento, tal que el segundo se presenta como especialización del primero y donde ambos involucran el concepto de dominio de protección:

- La primera posibilidad, extrapolable al resto de arquitecturas, se apoya de manera exclusiva en el uso de regiones para esta labor, esto es, cada espacio de pila se envuelve mediante una entrada completa de la unidad de protección o dicho de otro modo, la totalidad de la región constituye la pila. Como consecuencia de ello, las restricciones que impone la arquitectura en la creación de región extienden y aplican en la misma medida a la pila, condicionando aspectos tales como su tamaño, hecho que puede ocasionar un fenómeno de fragmentación interna, así como su posición de inicio y el alineamiento de esta. Puesto que cada tarea hace uso de al menos una región se evidencia una limitación en el número máximo de tareas –si y solo si se emplea un esquema estático para la representación, de manera que las regiones destinadas a este fin no son reutilizables y tan sólo se habilitan y deshabilitan según corresponda durante la planificación– y del mismo modo, dichas ventanas no podrán utilizarse para otros fines, limitando las posibilidades del sistema. No obstante, este esquema mejora las latencias en cambio de contexto al no sufrir modificación alguna en lo que respecta a sus propiedades y atributos. La limitación del crecimiento para la pila se consigue conteniendo ambos extremos mediante regiones, que no constituyen el punto de vista de la tarea y bien presentan los permisos de operación revocados o no resultan activas.
- Esta segunda opción considera algunas de las particularidades que se advierten para la arquitectura

ARM®, en concreto la división simétrica de una región en subregiones, propiedad que resulta, bajo ciertas circunstancias, una ventaja frente a aquellas microarquitecturas que tan sólo proveen de regiones atómicas para la protección. Dada su especificidad, el uso de subregiones exige de requisitos concretos para su aplicación, entre los que se encuentra la existencia de espacios contiguos en memoria con las mismas necesidades de atributos, extensión y privilegios y que pueden intercalarse en función de la tarea considerada como activa para un instante de tiempo. Tal y como se observa, esta descripción encaja con las demandas establecidas para la representación de los espacios de pila, de modo las regiones empleadas con este fin se sustituyen por una única ventana de protección, donde cada subregión se asocia a una pila de tarea. La primera ventaja que se obtiene de este hecho es un aumento del número total disponible de regiones, previamente ocupadas, que podrán destinarse a otras labores. Otra virtud observable corresponde a la mejora en rendimiento de cambio de contexto, en tanto en cuanto el número de registros de arquitectura empleado para la configuración se reduce por cada región liberada y en consecuencia las operaciones de acceso requeridas para su modificación, tal que pueden llevarse a cabo en única acción para este caso. Asimismo se reducirá el espacio empleado por la estructura de datos que describe cada ventana de protección asociada al punto de vista de una tarea. Sin embargo, esta elección no se encuentra exenta de desventajas. En concreto esta presenta una mayor fragmentación interna frente a la anterior propuesta, que consigue un mejor ajuste, condicionada por la característica de simetría, en donde el mínimo tamaño posible para una subregión determinará el del resto, aún a pesar de las diferentes necesidades de uso para aquellas tareas que hacen uso de la misma región. A pesar de ello, este efecto puede mitigarse mediante la combinación de secciones de código inmutables que comparten atributos y características de protección, con objeto de cubrir la totalidad del espacio dispuesto. Cada subregión queda delimitada por aquellas que la suceden y en el caso de los extremos, por el propio límite de región.

Puesto que los inconvenientes asociados a la última de las opciones son superados por sus beneficios, será esta la escogida como solución al problema de desbordamiento. Se propone con este fin un flujo que pretende simular la invocación recursiva de una función, logrando el mismo efecto que el ocurrido ante una situación de ejecución real. El momento exacto en el que se exceden los límites de pila se define tras alcanzar un valor como representación de la condición real, que indicaría el sorpaso de la profundidad máxima. Previamente dispuestas las correspondientes ventanas de protección y tras este hecho, se proyecta un acceso a una posición que carece de los correspondientes permisos, esto es, no se ajusta al dominio de la tarea activa. En reacción a este hecho, la unidad de protección lanzará una excepción de tipo *data abort*, interrumpiendo la ejecución en ese preciso instante. La señal se captura a través de la lógica interna de la plataforma, que invocará, de manera autónoma, la respectiva rutina de tratamiento, cuya finalidad se resume en informar aspectos tales como el punto de fallo o la tarea que propició el suceso, preparar el entorno para garantizar la continuidad y eliminar de la planificación dicha tarea. Tras ello se procede al cambio de contexto y restauración de la ventana de protección correspondiente a la nueva tarea, cuya existencia demuestra la continuidad de la plataforma en condición de excepción. Ambas tareas deberán notificar su ejecución mediante el uso de un *led* –salida/entrada genérica– y de un mecanismo de comunicación serie, en concreto la *UART*.

La figura 5.1 presenta la organización y despliegue de la lógica que conforma al sistema pergeñado, en condición de prototipo, así como aquellos aspectos correspondientes a la aplicación, en el denominado mapa de memoria del dispositivo. La distribución aquí planteada resulta aproximada, dada la variabilidad en lo que a construcción *software* se refiere, condicionada por aspectos de portabilidad definidos en el estándar de *C*. amplitud no se encuentra a escala –circunstancia que se extrapola al resto de diagramas incluidos bajo este capítulo–. La posición inicial alberga el vector de reinicio que contiene ocho entradas de 4*bytes* cada

una, en representación de las posibles excepciones para la arquitectura aquí empleada. El bloque adyacente dispone el grueso de la lógica relativa al sistema, seguida del *bootloader*, encargado de configurar el estado de ejecución conocido y viable. Tras estas de disponen los adaptadores para las llamadas de sistema y el código asociado a cada una de las tareas. Por último, se encuentra la información correspondiente tanto a objetos del sistema como de las tareas, bajo la denominada sección *.data*, cuyo contenido se traslada junto a la extensión de *.bss* al espacio de memoria principal, dado su carácter mutable. Cabe señalar que con objeto de maximizar la configuración de protección, así como prevenir comportamientos indefinidos, se negará el permiso de ejecución para dicho espacio, de manera que su contenido nunca podrá ser interpretado como instrucciones por el decodificador.

Tal y como se indica, se recoge una única región de protección para la representación de los espacios de pila. Esta se divide de manera simétrica en ocho fragmentos, de forma que uno de ellos se reserva para la realización de la pila de sistema, que se emplea en situaciones como la planificación, la provisión de servicios o la gestión de interrupciones. Mientras, el resto de subregiones se asignan a las tareas concurrentes, posibilitando un máximo de siete, aunque no se exige su uso completo si el número de tareas no resulta suficiente para agotarlas. Las mencionadas subregiones inactivas, aún a pesar de no cumplir una rol específico, sí cobran significado para la tarea activa, en cuanto que reflejan la prohibición de acceso sobre el espacio que encierran. Cabe señalar que dado que resulta inviable el solapamiento de las dimensiones ejecución, el diagrama no expresa los dominios de protección asociados a las tareas de usuario y sistema respectivamente para un momento concreto, sino que refleja los distintos valores para los atributos de la región, que dependerán del contexto en el que se encuentra la plataforma. Así pues, durante la ejecución de la tarea activa, la subregión asociada al *kernel*, que actúa como límite superior para el desbordamiento, permanece inactiva y por tanto innacesible, dado que el acceso para la modificación de la *MPU* se encuentra restringido. Una vez producido el cambio de modo, esto es, bien la tarea solicita un servicio o se produce un evento que debe ser tratado, es cuando se llevará a cabo la nueva configuración de protección. El primer paso consiste en intercambiar la subregión asociada a la pila del sistema y es en esta la misma operación –puede deducirse que esta aproximación minimiza el coste añadido para la restauración de espacios de pila, ya que al tratarse de la misma región puede combinarse las acciones– donde se inhabilita el espacio de pila asociado a la tarea activa, que pasará a formar parte del nuevo límite superior o inferior. Una vez determinada aquella tarea candidata a ejecutar, se configuran sus capacidades y el proceso que sigue, siendo el inverso al primer paso, restituirá el espacio para la tarea planificada, a la par que desactiva aquel correspondiente al *kernel*. Haciendo uso del solapamiento entre regiones y la prioridad en acceso se establecerá una región de fondo, como último elemento participante en el control del desbordamiento. Su finalidad es la de capturar cualquier operación sobre direcciones no contenidas bajo ninguna otra región y por tanto en el dominio de la tarea. Dicha región de fondo, cuya creación tiene lugar en el arranque de la plataforma y permanecerá inmutable durante la vida útil de la aplicación, se dispone en la totalidad del espacio de direcciones.

Cabe proponer que este planteamiento puede considerarse insuficiente en términos de corrección, en tanto en cuanto el espacio de pila del *kernel* comparte los permisos de lectura y escritura con el modo usuario. A efectos prácticos, dado que dicho modo restringe el acceso a la *MPU* para su configuración, la subregión *S0* correspondiente a la región *R3* se estima inaccesible, al permanecer inactiva durante la ejecución de la tarea. De esta forma, se encuentra un único punto crítico en el mecanismo frente a una disposición adecuada a las necesidades del modo supervisor, ya que tan sólo sería necesaria una perturbación en el registro que habilita dicha subregión para causar estragos a la estabilidad de la plataforma. Con todo, dada la remota probabilidad de que este hecho ocurra, la gravedad de este puede considerarse mínima. Su existencia sólo sería resultado de causas externas a la lógica contemplada del sistema –suponiendo la ausencia de vulne-

rabilidades y *bugs*– y que repercuten en la configuración de la *MPU*, por lo que la anterior propuesta para el control de desbordamiento puede considerarse suficiente. No obstante, se ofrece una alternativa que garantiza una mayor robustez, en cuanto que también involucra el registro asociado a la asignación de permisos y contempla la asignación de permisos adecuados para la representación de la pila de sistema, de utilidad bajo determinadas situaciones que serán tratas más adelante.

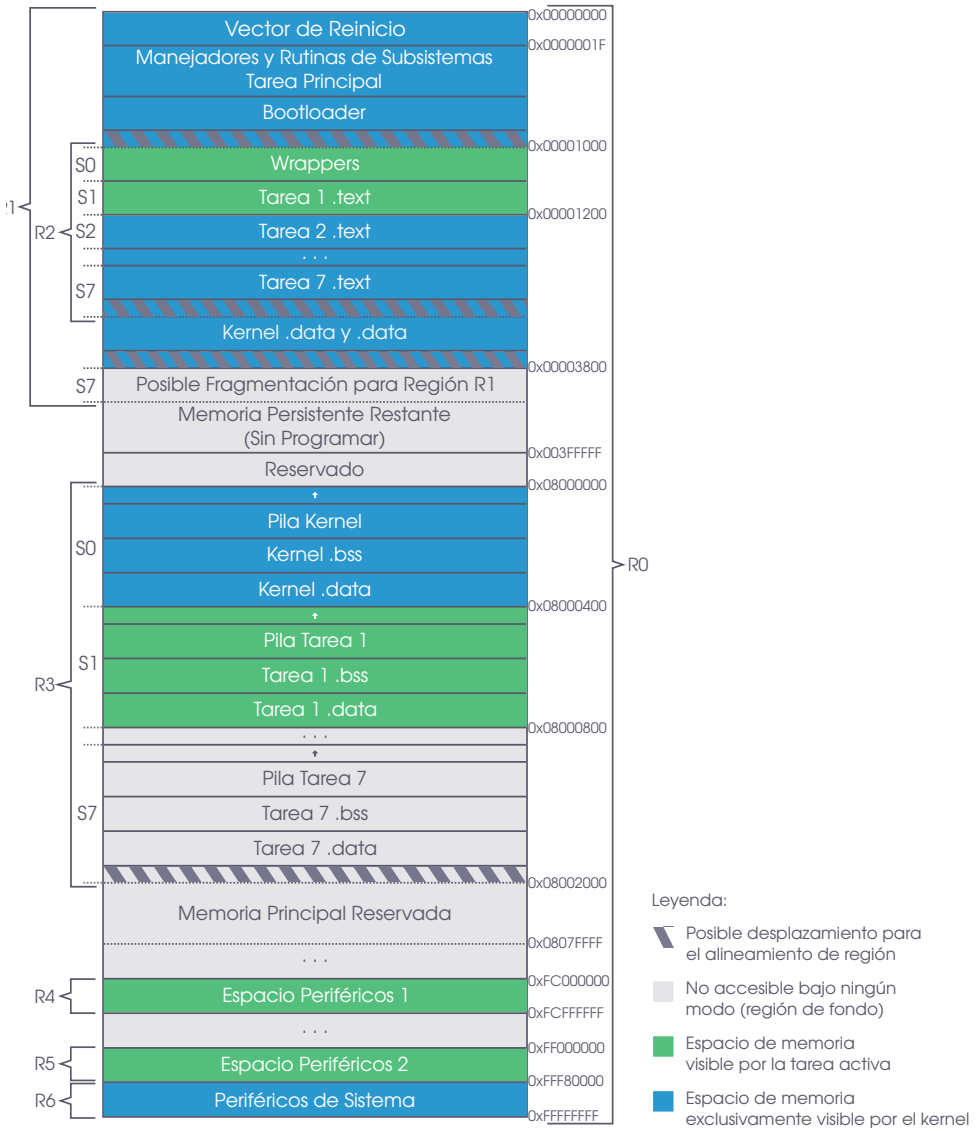

FIGURA 5.1: Representación del espacio de direcciones para una primera aproximación de los espacios de pila.

Así pues, bajo esta nueva propuesta, caracterizada por el diagrama 5.2, se opta por el uso de dos regiones para la comprobación en desbordamiento. En este caso, tan sólo se modificará la configuración para la ventana asociada a la representación de pilas de usuario, ya que aquella relativa al espacio del *kernel* permanecerá siempre activa con independencia de la dimensión, aunque sólo resultará accesible desde el modo privilegiado. Tal y como se ha comentado, existen dos situaciones por las que esta aproximación supone una ventaja frente a la anterior, en concreto para aquellos casos en los que la tarea planificada resulta la misma que la anterior tarea activa y la tramitación de excepciones para las que no existe razón para la conservación del estado previo de la tarea. Esta aceleración sólo es posible cuando no existe necesidad de alterar el punto de vista correspondiente a la tarea activa, dado que ya no se requiere restaurar el espacio de pila del sistema –siempre activo–. No obstante se trata de dos coyunturas excepcionales que exceden las competencias de este trabajo. Su inclusión tan sólo permite demostrar la existencia de diferentes concepciones para el control de desbordamiento de pila.

TABLA 5.1: Tabla de permisos y distribución correspondiente a la figura 5.1

Región	Subregiones Habilitadas	Dirección Base	Tamaño	Usuario	Privilegiado	Ejecución	Tipo de Memoria
0 (R0)	S0 - S7	0x00000000	4GiB	-	-	No	*
1 (R1)	S0 - S6	0x00000000	16KiB	-	R	Sí	Persistente (ROM)
2 (R2)	S0 - S*	0x00001000	2KiB	R	R	Sí	Persistente (ROM)
3 (R3)	S0 - S*	0x08000000	8KiB	RW	RW	No	Principal (SRAM)
4 (R4)	S0 - S7	0xFC000000	16MiB	RW	RW	No	Periféricos 1
5 (R5)	S0 - S6	0xFF000000	16MiB	RW	RW	No	Periféricos 2
6 (R6)	S0 - S7	0xFFF80000	512KiB	-	RW	No	Periféricos Sistema

Leyenda: **S*** - Cualquier subregión restante, **R** - Sólo lectura, **RW** - Lectura y escritura, **NA** - Sin acceso

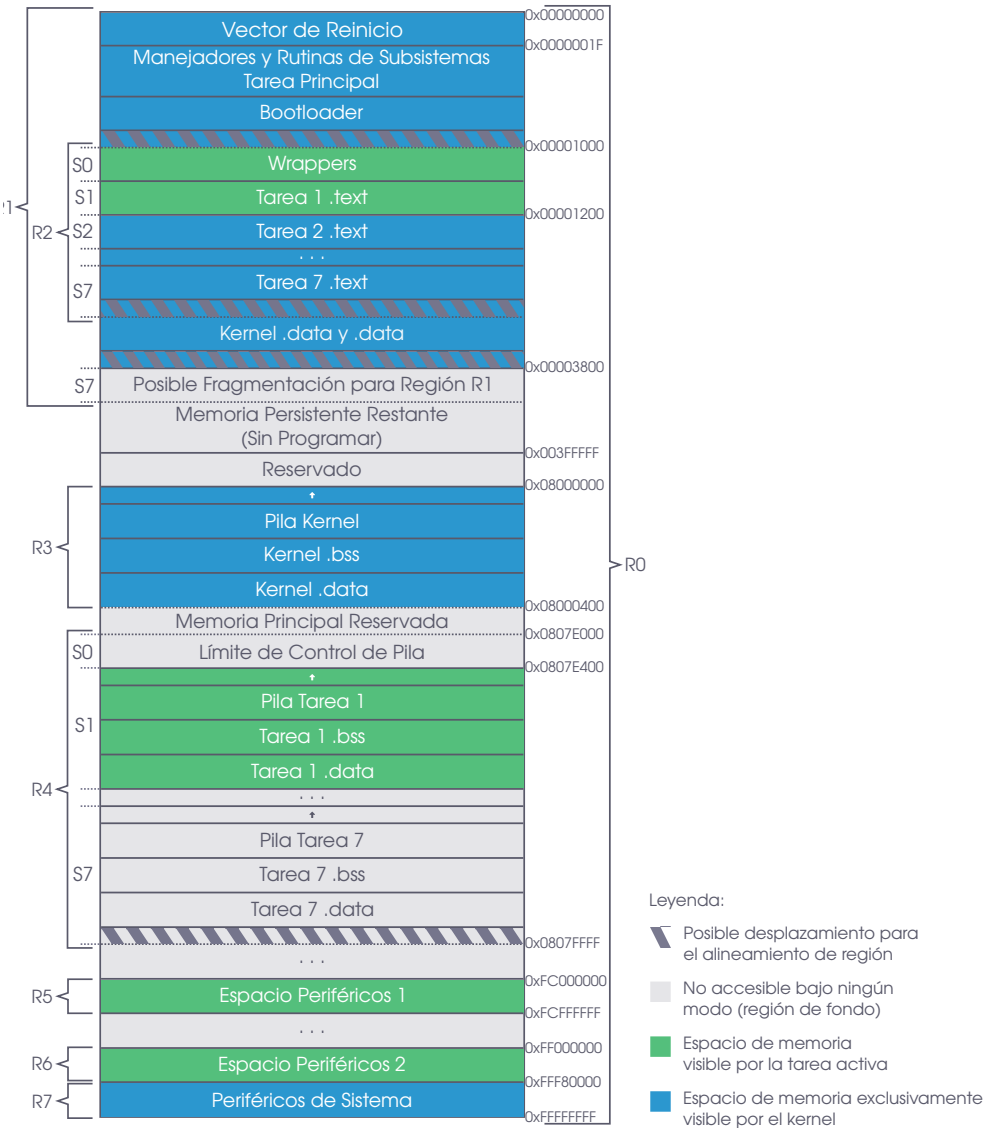

FIGURA 5.2: Segunda propuesta para la representación de los espacios de pila con protección en el desbordamiento.

Con esta implementación se alcanza así la independencia frente a esquemas de protección dinámicos y a aquellos ofrecidos por lenguajes de programación, para accesos destinados al rango de direcciones de la memoria principal. Tan sólo se necesitará conocer de antemano la información destinada a la representación del espacio de pila para cada tarea.

TABLA 5.2: Tabla de permisos y distribución correspondiente a la figura 5.2

Región	Subregiones Habilitadas	Dirección Base	Tamaño	Usuario	Privilegiado	Ejecución	Tipo de Memoria
0 (R0)	S0 - S7	0x00000000	4GiB	-	-	No	*
1 (R1)	S0 - S6	0x00000000	16KiB	-	R	Sí	Persistente (ROM)
2 (R2)	S0 - S*	0x00001000	2KiB	R	R	Sí	Persistente (ROM)
3 (R3)	S0 - S7	0x08000000	2KiB	-	RW	No	Principal (SRAM)
4 (R4)	S*	0x0807E000	8KiB	RW	RW	No	Principal (SRAM)
5 (R5)	S0 - S7	0xFC000000	16MiB	RW	RW	No	Periféricos 1
6 (R6)	S0 - S6	0xFF000000	16MiB	RW	RW	No	Periféricos 2
7 (R7)	S0 - S7	0xFFF80000	512KiB	-	RW	No	Periféricos Sistema

Leyenda: **S*** - Cualquier subregión restante, **R** - Sólo lectura, **RW** - Lectura y escritura, **NA** - Sin acceso

5.2 EJECUCIÓN ESPECULATIVA

A lo largo de este apartado, se tratará de evidenciar cómo la unidad de protección de memoria permite mitigar comportamientos no triviales, a menudo presentes de forma particular en las microarquitecturas y por extensión en la plataforma subyacente elegida para el desarrollo. Previo paso se introducen sumariamente unas nociones teóricas como contexto al problema tratado.

Debido a la existencia de separación entre dimensiones de ejecución se requiere de un mecanismo para el escalado de privilegios, tal que permita llevar a cabo la acción de un servicio perteneciente al modo privilegiado. Comúnmente este proceso se denomina como llamada al sistema y para el que, dada su relevancia, los proveedores de arquitectura facilitan ua instrucción específica que consigue este efecto a través de la generación de una excepción. *ARM*®ofrece una operación exclusiva denominada llamada a modo supervisor o *SVC* (*SuperVisor Call*). Para lograr su acción, esta deberá ir acompañada de un código que identifica el correspondiente servicio. La elección de la rutina adecuada para el tratamiento dependerá del esquema empleado, siendo aquella aproximación que distingue entre rutinas de servicio diferidas (*DFSR*) y rutinas de servicio (*ISR*) la empleada en este trabajo, dado que disminuye la carga para el procesamiento del evento. Una vez se establece la rutina diferida correcta para el tratamiento y esta completa su actividad, entonces es necesario transferir el control a la tarea que en primer lugar solicitó dicho servicio. *ARM*®especifica la instrucción *"MOVS PC, R14_svc"* con este fin, donde *R14_svc* contiene la dirección inmediatamente siguiente al punto en el cual la tarea generó la llamada. Sin embargo, esta instrucción presenta una contrapartida, puesto que no permite regresar de manera directa a una posición anterior a aquella de retorno por defecto, lo que implica la ejecución de una operación previa a esta con objeto de ajustar correctamente el valor de destino deseado. Por este motivo, la arquitectura escogida también ofrece una alternativa, de la forma *"SUBS PC, R14_svc, #n"*, que combina los efectos de estas dos acciones en una única instrucción equivalente. El sustraendo ***n*** indicará la cantidad de *bytes* a desplazar previa a la posición identificada por *R14_svc*. Cabe señalar que esta última solución no se limita únicamente a la invocación de servicios, sino que puede emplearse para cualquier retorno que conlleva un cambio al modo usuario. Asimismo, sufre de un particular efecto que resulta de interés y será objeto de estudio en lo sucesivo.

Dada la discrepancia en ciclos de acceso para los distintos tipos de memoria dispuestos en la plataforma –aumentan las latencias a medida que se ascienden en la jerarquía– se introducen técnicas que mantienen una elevada tasa de ocupación en la unidad de procesamiento. Por ello, la microarquitectura habilita la de-

codificación de instrucciones en bloque, que permanecerán a la espera en un *búfer* de hasta tres entradas, mediante la *prefetch unit* que se encarga de acceder al contenido de la posición que codifican como destino previo paso a su ejecución. Ante la aparición de bifurcaciones en la lógica, se producen ramificaciones que impiden la correcta aplicación de la técnica mencionada. La búsqueda de las operaciones esperadas para el nuevo flujo ocasiona que la unidad de procesamiento permanezca ociosa durante al menos varios ciclos, hecho que se suma a la secuencia de instrucciones previamente decodificadas, que quedan descartadas, acrecentando aún más la pérdida de rendimiento. De nuevo y para solventar esta situación, la arquitectura introduce otro mecanismo, en este caso para la predicción de bifurcaciones, encargado de dilucidar el camino correcto dada una condición de salto –en ausencia de esta la bifurcación siempre se toma–. Esta información se transmite a la *prefetch unit*, que continúa de esta manera en el punto esperado. Si se considera una bifurcación sin condición para el retorno en tratamiento de excepción, entonces es de esperar que la ejecución continúe a partir de la dirección de destino. Sin embargo, la instrucción *"SUBS PC, R14_svc, #n"* presenta una comportamiento anómalo, ya que pese a no incluir condición alguna en su representación, no impone ninguna barrera a la ejecución especulativa, de modo que la decodificación prosigue secuencialmente y por tanto el contenido del *búfer* se considera incorrecto.

Ambos mecanismos descritos para la ejecución especulativa, entroncan con dos características presentes en la plataforma escogida para el prototipado, en concreto, la presencia de memoria –persistente y principal– con soporte *ECC* y de un mecanismo para la captura de eventos y notificación de errores propietario, denominado módulo de notificación de errores –*Error Signaling Module* o *ESM*–. La primera de ellas se destina a la integridad de señales que hacen uso del bus del sistema, entre las que se incluyen los accesos a las jerarquías de memoria, de forma que se genera un evento en caso de detectar una perturbación. Cabe señalar que la microarquitectura especifica la circunstancia que da origen a un error [5], hecho que ocurre únicamente cuando la operación que provocó el evento completa su ejecución en la unidad de procesamiento. Esta acción queda reflejada en un cojunto de registros específicos de la arquitectura, con objeto de identificar la causa del error y revertir su efecto. Mientras, los eventos resultan visibles bajo un bus de la arquitectura denominado *EVNTBUS**m***, cuya salida se encuentra conectada a la segunda de las características, esto es el *ESM*, donde su función consiste en clasificar las señales recibidas y establecer para cada una de ellas niveles de gravedad. ASí pues, un evento no corregible en comprobación de *ECC* se evalúa como crítico, por lo que es preciso su tratamiento, hecho que se comunica mediante una interrupción *FIQ* –en términos de la arquitectura– y de forma paralela a través de una interfaz a la que denomina salida de error, conectada en el caso de la plataforma a una interfaz de entrada/salida genérica. Si se atiene a este mismo comportamiento esta vez para un acceso especulativo generado por la *prefetch unit* y cuya operación nunca procede –se descarta por cambio en el flujo, tal y como se describe para la codificacion *SUBS*–, entonces se producirá igualmente una interrupción que tendrá lugar tras completar la transferencia de control a la tarea de usuario. El siguiente paso lógico sería tratar dicha interrupción, pero puesto que no queda constancia de su efecto, en tanto en cuanto no se origina un error –la acción nunca llega a realizarse–, no es posible determinar su causa. Se produciría así un fenómeno de falso positivo, hecho de gravedad si se tiene en consideración la integración del sistema embebido en un sistema anfitrión. Por ejemplo, si la salida del error se encuentra conectada a una entrada del anfitrión para la comunicación, este segundo podría paralizar o bloquear su ejecución a la espera de una resolución que no llegaría nunca, al no existir tal. La aparición de este tipo de circunstancias anómalas se ha comprobado frecuente durante las fases iniciales de la codificación, en tanto en cuanto se lleva a cabo una programación de la memoria persistente de forma continuada, hecho que incrementa la aparición de secciones con valores de *ECC* no actualizados correctamente –a causa de la posible variación de tamaño del *binario* empleado–.

Tal y como se observa, la figura 5.3 ejemplifica la circunstancia anteriormente descrita. La subrutina aquí

FIGURA 5.3: Fragmento de código que recoge el fenómeno bajo estudio.

```
00000044 <_svcdummy_>:
  44:   e92d4003        push    {r0, r1, lr}
  48:   e30fe524        movw    lr, #62756      ; 0xf524
  4c:   e34fefff        movt    lr, #65535      ; 0xffff
  50:   e59e1000        ldr     r1, [lr]
  54:   e28f0028        add     r0, pc, #40     ; 0x28
  58:   ebffffe         bl      0 <_printf>
  5c:   e59f019c        ldr     r0, [pc, #412]  ; 200 <TASKPC+0x29>
  60:   e5901000        ldr     r1, [r0]
  64:   e3510003        cmp     r1, #3
  68:   e2811001        add     r1, r1, #1
  6c:   0bffffe         bleq    0 <_gotoCPUIdle>
  70:   15801000        strne   r1, [r0]
  74:   e8bd4003        pop     {r0, r1, lr}
  78:   e25ef014        subs    pc, lr, #20
  7c:   ea001027        b       4120 <TASKPC+0x3f49>
```

mostrada se encarga de procesar una llamada de sistema, por lo que existe un cambio de modo previo paso a su invocación, así como en el retorno. Se puede apreciar la codificación de la instrucción *SUBS* en la línea número 0x78, seguida del valor 0x0ea001027 y decodificado como una instrucción de bifurcación. Dado que *SUBS* no presenta ningún tipo de condición –por ejemplo sufijos -ne o -eq–, entonces puede deducirse que el mecanismo de prediccion de bifurcaciones marcará el retorno tomado, continuando la ejecución en la dirección resultado de la diferencia del registro *lr*. Sin embargo, no ocurre este hecho, por lo que la *prefetch unit* decodificará el contenido indicado en la línea 0x7C, tomando el salto y accediendo especulativamente a la posicion codificada como destino. Si el contenido en ese punto no presenta los valores correctos de *ECC*, entonces tendrá lugar el mencionado falso positivo, ya que realmente el flujo continuará en la correspondiente tarea de usuario. Con intención de prevenir este fenómeno se proponen varias soluciones entre las que se involucra aquella que hace uso de la unidad de protección de memoria:

- La primera solución plantea el uso de un relleno o *padding*, mediante instrucciones cuya ejecución no produce efecto alguno (*nop*), para el espacio no considerado durante la programación del código de aplicación. Se consigue así el efecto deseado en el que cualquier acceso no previsto queda sin influencia, puesto que los valores *ECC* para dichas posiciones quedan correctamente asignados. Sin embargo, esta alternativa plantea dos inconvenientes, en concreto:
  - Esta aproximación no resulta suficiente para atajar la posibilidad de decaimiento en los valores del contenido de la memoria, que afecta tanto a contenido considerado de confianza como a aquellas secciones críticas, puesto que no impide la ocurrencia de un acceso –directo o especulativo– a posiciones con contenido corrompido. Dicha mutación puede deberse, dada la existencia de interacción entre el entorno y la plataforma, a bien la degradación de componentes físicos o por influencia electromagnética.
  - La continuidad de un sistema embebido resulta de gran importancia para garantizar el correcto funcionamiento, la seguridad y estabilidad de los sistemas y entornos en los que estos se agregan, por lo que será de obligado cumplimiento maximizar su disponibilidad. Existen sistemas embebidos no accesibles una vez son desplegados, pero que requieren renovar su comportamiento,

incrementando así su funcionalidad e incorporando nueva información sobre su ámbito durante su tiempo de vida. Una actualización del *firmware* requiere la programación de la memoria de la plataforma, lo que conlleva por otra parte la correcta asignación de valores *ECC*, cuya tecnología subyacente impone en ocasiones restricciones sobre el rango determinado para la operación. Por ejemplo, no sólo se programa el correspondiente tamaño del *binario*, sino que comprende unidades de espacio fijas denominadas como sectores, que pueden superar con creces la extensión del *firmware*. La inclusión de un relleno para la protección puede ocasionar el desbordamiento de uno de estos sectores, lo que supondría una sobrecarga a dicha operación de programación, en tanto en cuanto se requiere ampliar el *padding* hasta cubrir el siguiente bloque. Asimismo, esta acción conlleva un aumento del tamaño del *binario*, hecho que influye directamente en la transmisión del *firmware* al dispositivo, ya que a mayor tamaño mayor tiempo tomado, situación que se agrava para aquellos casos en los que este resulta innacesible.

- Puesto que la anterior opción se presta insuficiente para los objetivos de la protección y por tanto de la estabilidad de la plataforma, se presenta una segunda alternativa con intención de corregir las limitaciones anteriores, valiéndose de la *MPU* para ello. La idea principal detrás de esta aproximación consiste en prevenir la aparición de la señal incorregible, dado cualquier acceso con una posición de destino para la que se desconoce su contenido. Se establecerán ventanas de protección que cubren aquellas extensiones del espacio sin programar, de modo que cualquier intento de manipular o leer un punto contenido en la región resulta denegado por defecto. En esta solución no se necesita rellenar por completo el espacio de direcciones, sino que la introducción de un desplazamiento responde a las restricciones de tamaño y posición de región que la arquitectura elegida presenta. Como consecuencia de este hecho, el sistema adquiere una mayor disponibilidad durante una actualización, ya que no existe motivo para ampliar el espacio programable, con la sobrecarga asociada que esto supone.

Tal y como se indica, esta segunda solución será aquella elegida para el tratamiento del fenómeno anteriormente documentado. Con objeto de probar esta técnica se pretende simular la condición de evento para error no generado mediante una aproximación de similares características al flujo incluido en la figura 5.3. Una posible propuesta a este respecto involucra la ejecución de una tarea de usuario que solicita un servicio de sistema, de forma que se produce un cambio de modo y por extensión fuerza la operación de retorno comentada. Este servicio resulta una simple maqueta sobre la que se inserta, tras la instrucción *SUBS* encargada de devolver la plataforma a su anterior estado, una palabra –en el sentido informático– cuidadosamente elegida, tal que codifica una rutina de bifuración, cuya posición de destino no dispone de los valores *ECC* adecuados. Una vez la tarea de usuario recupera el control de la ejecución, esta se encargará de comprobar el estado del módulo *ESM*. En caso de existir un cambio, esto es, se captura la señal generada en *EVNTBUS**m***, entonces se configura convenientemente la unidad de protección, excluyendo cualquier espacio no asociado a la aplicación. Tras ello, se reproducirá el proceso aquí descrito una segunda vez, con la intención de demostrar que en efecto, no se visualiza ningún evento asociado al acceso especulativo, lo que prueba la eficacia de la *MPU* para esta tarea.

La figura 5.2, previamente empleada en la descripción del desbordamiento, ejemplifica igualmente la representación de la solución aquí propuesta. Con independencia de la distribución de los dominios tanto de usuario como de *kernel*, se dispone una región de fondo que niega cualquier acceso posible para la totalidad del espacio. De esta manera cualquier acceso directo o especulativo para espacios no contemplados no surtirá efecto alguno al rechazarse la operación misma. No obstante, conviene aclarar que la técnica especificada no cubrirá aquella casuística que afecta a secciones de código o datos confiables, dado que un cambio en el contenido de cualquier posición contenida en estas deberá ser convenientemente tratado, puesto que supone una amenaza real a la estabilidad de la plataforma. No obstante dicha figura presenta un potencial

riesgo que entronca con la ejecución especulativa descrita, de modo que se considera conveniente ofrecer una alternativa más correcta en cuanto a protección se refiere.

En concreto, al emplear una única región para la protección de la memoria persistente, en lo que a dominio del sistema se refiere, todos los privilegios y atributos de región aplican por igual a las distintas secciones que constituyen la aplicación. Por extensión, aquellas empleadas para la representación de datos adquirirán los correspondientes permisos de ejecución. Este hecho implica que la *prefetch unit* podría interpretar cualquier valor contenido en dicho espacio como si de instrucciones se tratasen, lo que arrastra consigo un elevado riesgo –no sólo en términos de seguridad, donde un atacante podría encontrar un flujo deseado en dicha sección mediante el empleo de una técnica conocida como *ROP*– puesto que en caso de producirse una condición anormal de flujo, sus consecuencias podrían magnificarse, con efectos desconocidos en la plataforma y en su entorno, si tiene como destino esta sección. Para evitar esta circunstancia se presenta, tal y como refleja la figura 5.4, una versión mejorada de las anteriores consideraciones. En ella se establece, tal y como demuestra 5.3, una región adicional para la que tan sólo se conceden los permisos de lectura, revocando aquellos asociados a la ejecución. Cabe señalar que la introducción de esta nueva región puede ocasionar una mayor fragmentación interna en la forma de desplazamientos para su correcto alineamiento. Puede deducirse así una relación directa para la tríada seguridad/estabilidad, tamaño del binario y número de ventanas de protección empleadas, de modo que cualquier aproximación deberá conseguir, como solución de compromiso, un balance entre características.

TABLA 5.3: Tabla de permisos y distribución correspondiente a la figura 5.4

Región	**Subregiones Habilitadas**	**Dirección Base**	**Tamaño**	**Usuario**	**Privilegiado**	**Ejecución**	**Tipo de Memoria**
0 (R0)	S0 - S7	0x00000000	4GiB	-	-	No	*
1 (R1)	S0 - S6	0x00000000	16KiB	-	R	Sí	Persistente (ROM)
2 (R2)	S0 - S*	0x00001000	2KiB	R	R	Sí	Persistente (ROM)
3 (R3)	S0 - S*	0x00001800	2KiB	-	R	No	Persistente (ROM)
4 (R4)	S0 - S7	0x08000000	2KiB	-	RW	No	Principal (SRAM)
5 (R5)	S*	0x0807E000	8KiB	RW	RW	No	Principal (SRAM)
6 (R6)	S0 - S7	0xFC000000	16MiB	RW	RW	No	Periféricos 1
7 (R7)	S0 - S6	0xFF000000	16MiB	RW	RW	No	Periféricos 2
8 (R8)	S0 - S7	0xFFF80000	512KiB	-	RW	No	Periféricos Sistema

Leyenda: **S*** - Cualquier subregión restante, **R** - Sólo lectura, **RW** - Lectura y escritura, **NA** - Sin acceso

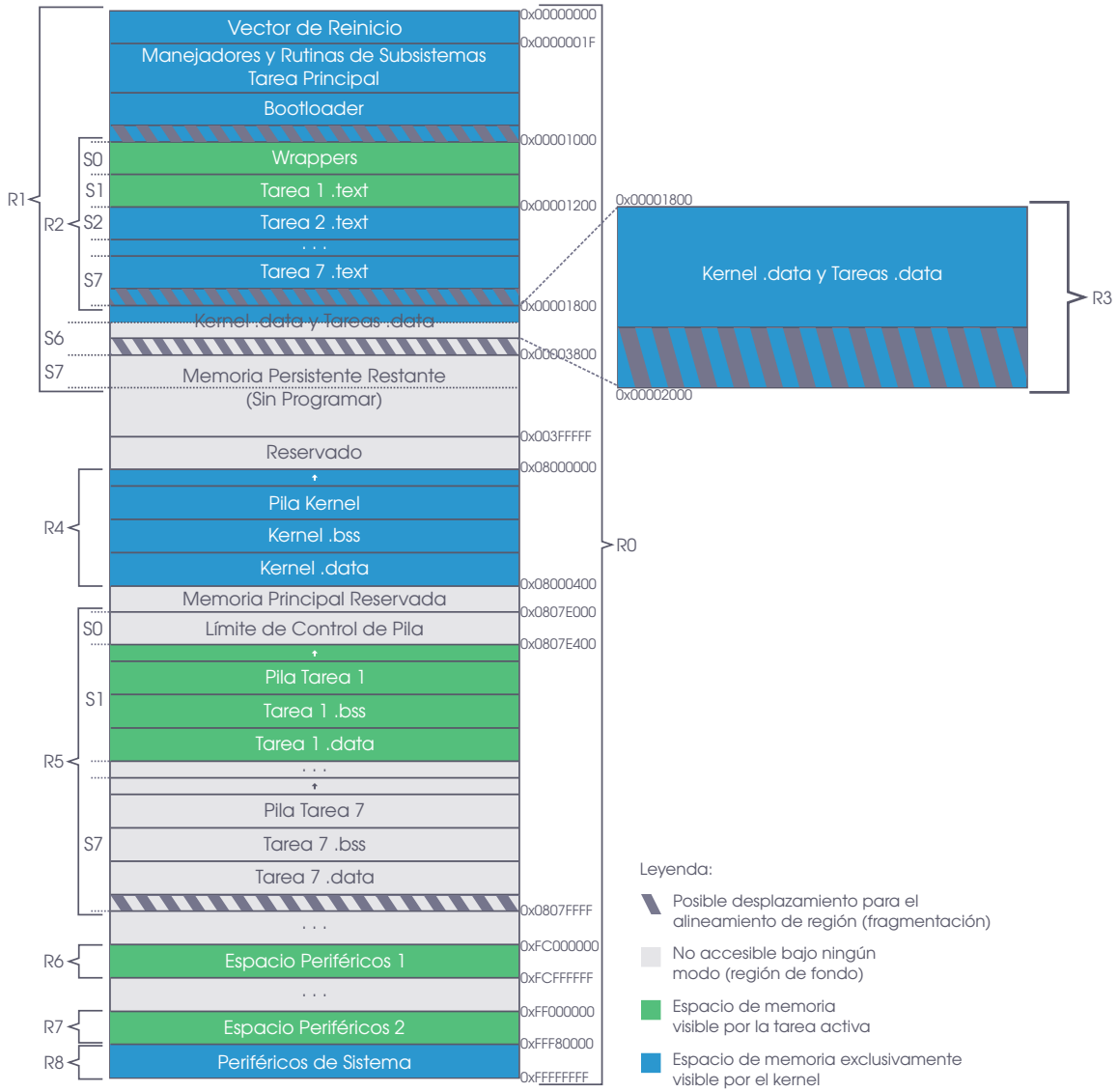

FIGURA 5.4: Propuesta que revoca los permisos de ejecución para las secciones de datos.

5.3 PLAN DE PRUEBAS

Para este apartado se delimitan los pasos esperados para la prueba y que seguirán las tareas provistas para la comprobación de los casos en estudio, así como las bondades de la unidad de protección. Estos permiten definir un estado conocido para el análisis de la propuesta, de manera que cualquier desviación deberá ser correctamente tratada.

5.3.1 DESBORDAMIENTO DE PILA

A continuación se lista los pasos esperados en la prueba para el caso de estudio que involucra el desbordamiento del espacio de memoria y con falta de permisos:

1. Se muestra tras el arranque un menú que permite seleccionar la opción a probar, donde en este caso corresponde con la primera de las posibilidades.

2. Se procede a la creación de ambas tareas, hecho que será mostrado por la interfaz serie, dando cuenta del tamaño de pila, así como el punto de entrada para cada una.

3. Tras el paso anterior, se continúa al modo de bajo consúmo energético, no sin antes permitir la captura de eventos de tiempo real.

4. La ejecución procederá a la planficación de tareas tras haberse producido la primera de las interrupciones programadas, de forma que ambas se turnan, en el acceso a los recursos de la plataforma. Estas notificarán de su ejecución mediante una interfaz genérica conectada a un *led*.

5. Al alcanzar la condición de simulación de desbordamiento, se procede al acceso indebido, hecho que deberá aparecer debidamente notificado en la interfaz de salida.

6. Este flujo de prueba continúa con el tratamiento del mencionado error. En este se comprobará que el tipo de evento contenido por el registro *DFSR*, al tratarse de un acceso a datos y no debido a la ejecución de una instrucción, se refiere al acceso con permisos insuficientes o un valor *0xD* en hexadecimal.

7. El resultado del tratamiento inhabilitará la tarea causante del error, eliminando a esta de la planificación y mostrando su información asociada.

8. La prueba concluirá con la restauración de un estado estable para la ejecución de aquella tarea todavía planificable, hecho evidenciado por el parpadeo de un único *led*.

Si los pasos anteriormente descritos proceden correctamente, entonces la prueba se lleva a cabo sin sobresaltos.

5.3.2 Memoria especulativa

Bajo el siguiente subapartado se establecen los pasos previstos durante la prueba para el caso de estudio en el que se produce un acceso especulativo:

1. Se muestra tras el arranque un menú que permite seleccionar la opción a probar, donde en este caso, esta se corresponde con la segunda.

2. Se procede, al igual que para la anterior propuesta, a la creación de dos tareas, hecho que será indicado a través de la interfaz serie, dando cuenta del tamaño de pila, así como el punto de entrada para cada una ellas.

3. Tras el paso anterior, se continúa al modo de bajo consúmo energético, no sin antes permitir la captura de eventos de tiempo real.

4. La ejecución procederá a la planficación de tareas tras haberse producido la primera de las interrupciones programadas, de forma que ambas se turnan, dada su prioridad, en el acceso a los recursos de la plataforma. Asimismo, su ejecución será señalada mediante el uso de la interfaz genérica conectada a un *led*.

5. Tras logar la condición para la simulación, se procederá al lanzamiento del desencadenante de ejecución especulativa, hecho que deberá ser debidamente comunicado.

6. Si el anterior paso se completa sin incidentes, entonces el evento de falso positivo en condición de error deberá aparecer mediante la notificación de un led, conectado a la salida de fallos. De igual modo, la ejecución procederá a partir del manejador de interrupción correspondiente.

7. En este punto se recibirá la información asociada a la tarea en ejecución para la que se registró tal condición, así como el contenido de los registros de error asociados a la *MPU* y las banderas de módulo de captura de señales. Como salida esperada, ninguno de los registros de estado de la unidad de protección deberá registrar contenido alguno.

8. Como paso final, la prueba concluye tras restaurar el anterior estado de ejecución de la plataforma, de modo que continúan ambas tareas, estableciendo previamente la región de protección adecuada, de modo que el evento estudiado no se verá reproducido en sucesivas iteraciones.

De nuevo, si los pasos anteriormente descritos proceden correctamente, entonces la prueba se lleva a cabo sin sobresaltos.

CAPÍTULO 6

CONCLUSIONES Y LÍNEAS DE TRABAJO FUTURAS

Como cierre a este trabajo y fruto de su elaboración se relatan algunas dificultades y contratiempos encontrados a lo largo del proceso, los cuales resultan de interés ya que permiten mejorar las técnicas y aproximaciones empleadas, así como las tecnologías utilizadas para futuros desarrollos o para aquellos con características comunes. Asimismo, se recogen algunas conclusiones derivadas y se incorporan algunas líneas de desarrollo futuro, con afán de indagar y profundizar en el ámbito de la estabilidad de los sistemas embebidos, de modo que quede evidencia de la variedad y extensión aún disponible para este área.

6.1 CONCLUSIONES

6.1.1 RECAPITULACIÓN

Tal y como se observa, la *MPU* es la técnica de protección preferida entre los fabricantes de microcontroladores, frente a soluciones presentes en sistemas de cómputo general o aquellas personalizadas, dado su equilibrio entre efectividad en su tarea y coste para su realización. Es posible encontrar múltiples arquitecturas, cada una con su propia aproximación para el mecanismo, que convergen en puntos comunes expuestos bajo las denominadas operaciones primitivas. Su uso destaca, en la medida en que consigue evadir imposiciones propias de soluciones dinámicas, que dependen en gran medida de los mecanismos de protección ofrecidos por el lenguaje y las herramientas de construcción de *software* escogidos –con la inconsistencia que esto en ocasiones supone– y sobrecargan la unidad de procesamiento para cada operación de acceso a memoria realizada. A priori, todo parece indicar un amplio soporte y fuerte presencia en sistemas operativos de tiempo real, destinados generalmente al ámbito de los sistemas embebidos, pareja a su existencia en microcontroladores, pero como apuntan estudios como [64] no resulta tal, ya que en ocasiones el soporte no es obligatorio o es parcial –cabe destacar la inclusión de *Zephyr* [65] y *RTEMS* [66] – sistemas operativos de tiempo real con licencia de código abierto –no recogidos bajo dicho estudio y que resultan una alternativa de interés, en cuanto que sí incluyen un mayor soporte para varias arquitecturas en este aspecto–. Si tan sólo se desea llevar a cabo el estudio de las bondades de estos mecanismos de protección, previo a su aplicación en un proyecto real, la adaptación de sistemas existentes a la plataforma elegida con este fin, supone una elevada inversión en recursos –las capacidades que estos sistemas ofrecen son proporcionales al incremento en uso de los medios disponibles– y tiempo no permisibles, y desperdiciados si la unidad de protección no supera siquiera la fase de evaluación. A este hecho se suma la lenta adopción de arquitecturas y mecanismos de última generación en ámbitos de aplicación comercial, asociados a aspectos tales como la estabilidad y seguridad de la plataforma –por ejemplo, la comprobación de límites de pila [67], alternativa que consigue reducir la sobrecarga de operación sobre la unidad de protección–, por motivos que abarcan desde reducir los costes de licencia de las tecnologías empleadas, hasta la incompatibilidad con el *software*

previamente desarrollado.

Dadas estas circunstancias y debido a la creciente complejidad que alcanzan los desarrollos de productos *software* en la actualidad –junto a los riesgos internos y externos que los amenazan–, se pone de manifiesto la importancia del desarrollo de un *protokernel* que albergue un subsistema para la gestión de la unidad de protección y sobre el que se ejecutarán tareas sencillas, cuyo propósito es exponer las ventajas y el rango de utilidad, que dicho mecanismo de protección puede ofrecer en el ámbito de los sistemas empotrados, mejorando así su adopción. Es mediante la realización de este sistema que se deducen los usos de la unidad de protección, los cuales comprenden desde la sincronización para el acceso compartido entre tareas concurrentes, el tipo de memoria accedido –implicaciones en el rendimiento de la operación–, el control del espacio destinado a la pilas de tarea, la prohibición de acceso directo a un periférico requiriendo así un servicio del sistema, hasta casos límites no triviales, cuyo impacto tanto en su entorno como en la propia plataforma se considera de gravedad.

No obstante, junto a las virtudes también se evidencian limitaciones, determinadas bien por su utilización o por características intrínsecas a su diseño. En lo relativo a su aplicación, la ejecución presenta, en el mejor de los casos, operaciones adicionales para cada cambio de contexto, mientras que en el peor escenario, esto es, en condición de excepción, el impacto es mayor, en cuanto que se requiere su tratamiento previo paso a dicho cambio. Dada la severidad que implica la existencia de un acceso no contemplado en primer lugar, así como el potencial resultado por su acción–la plataforma o el sistema en el que se embebe, quedan inoperantes o en el peor de los casos puede ocurrir un perjuicio físico de cualquier índole para el ámbito de aplicación–, cualquier posible reducción en el rendimiento de la aplicación se considera asumible. Respecto a las particularidades de la unidad de protección, estas determinan el resultado de la implementación –por ejemplo, las limitaciones que imprime la arquitectura y/o microarquitectura en la región de protección –dirección de inicio, tamaño o dirección de fin —, lo que evidencia la inviabilidad de ignorar la plataforma subyacente, dado que también se extiende su influencia a la realización de un producto, esto es, su desarrollo, aunque metodologías como la escogida para el desarrollo, permiten por contra reducir dicha influencia. Destaca dada su importancia, además de las ya mencionadas limitaciones, el número de regiones de protección disponibles, que por su carácter finito hace necesario el compromiso entre la capacidad de protección para el sistema, el número máximo de tareas disponibles y los tiempos de cambio de contexto –cuantas más regiones se emplean para la representación de un punto de vista de tarea, mayor será el tiempo necesitado para su intercambio –aunque existen excepciones, como demuestra *TriCore*–. El impacto de dichas restricciones condicionan la ventajas y desventajas que presentan las diferentes implementaciones del mecanismo, dado un caso de aplicación concreto, de forma que encontrar la solución perfecta resulta improbable, hecho que demanda del uso de heurísticas, con objeto de obtener configuraciones consideradas como adecuadas y maximizar la eficacia que presenta la unidad de protección en su tarea.

Por último, cabe destacar que la aproximación para el desarrollo elegida –orientada a modelo– resulta de ayuda con motivo de reducir el peso que la arquitectura impone sobre el producto. En detrimento, esta limita la elección de herramientas *CASE* para la que demanda soporte tanto del estándar empleado en la representación de diagramas, como de la capacidad para la transformación automática de modelos, aunque como se evidencia, es posible postergar esta última mediante conversión manual. Del mismo modo, aquellas que conforman el entorno de construcción son condicionadas, bien por la arquitectura escogida o por la implementación de fabricante seleccionada para el dicha arquitectura y que constituye el *hardware* subyacente. Puesto que todas las herramientas aquí empleadas son en sí mismas productos *software*, estas se encuentran sujetas a defectos que las vuelven inoperantes o cuya acción produce resultados inconsisten-

tes para diversas iteraciones. Además, la portabilidad del código resultante, no es un factor determinado en exclusiva por las particularidades de la plataforma propuesta, sino que también depende de aquellas propiedades que define el estándar para el lenguaje de programación escogido, hecho cuyo efecto extiende a la implementación de la herramienta de construcción *software* que soporta dicho lenguaje.

6.1.2 Análisis de adversidades identificadas

El siguiente apartado relata, dada su relevancia, algunos de los problemas encontrados a lo largo del desarrollo, en tanto en cuanto determinan cómo se aborda el proceso. Se pretende establecer una guía que compendie todas estas adversidades, de cara a afrontar futuros desarrollos para el ámbito de aplicación de los sistemas embebidos. Esta referencia consigue reducir el tiempo empleado para dilucidar una solución en presencia de cualquier contratiempo. Asimismo permitirá perfeccionar y dominar técnicas y herramientas de desarrollo a emplear, en la medida en que puede mitigarse la aparición o en su defecto corregirse errores que ocurren durante su utilización.

La primera de las cuestiones aquí planteadas, vendría determinada por la plataforma escogida para la realización de este trabajo, como implementación de la microarquitectura seleccionada. En concreto, la elección es la del dispositivo *TMS570LC4357*, recomendado según el fabricante –*Texas Instruments* [68]– para el campo de la automoción, en el que se requiere la aplicación de normas tales como *ISO 26262* [69] o *IEC 61508* [70] (esta última de interés aunque no se aborda en este documento). Las plataformas concebidas para este ámbito de aplicación, presentan mecanismos de redundancia y control, que facilitan la consulta del estado de ejecución, así como numerosos periféricos para la captura y notificación de condición de estado. Este tipo de módulos lógicos son acoplados a la arquitectura por el propio fabricante (alguno en términos propietarios –esto es módulos *IP*–, lo que requiere acudir a los manuales de referencia facilitados por este), de entre los que destaca el mecanismo de interconexión, hecho que implica una interdependencia con la lógica de procesamiento. Por consiguiente, para la correcta operación de la plataforma, necesariamente se requiere de la configuración e inicio de estos elementos, operación que además, se acompaña del lanzamiento de pruebas de diagnóstico específicas, cuya finalidad es asegurar que siguen en funcionamiento y no presentan algún defecto. Todo ello aviva la necesidad de dominar los distintos módulos, de manera que surge una doble complejidad que se refleja, por un lado en el aprendizaje de las posibilidades capacidades de los componentes dada su variedad y extensión, siendo estos en algunos casos además oblicuos al tema central aquí tratado (unidad de protección de memoria) y que como tal no aportan valor directo, como por otro, en el desarrollo del sistema aquí provisto, que deberá recoger la lógica que rige a estos mecanismos (de manera que se aplican los conocimientos adquiridos) y que tiene como finalidad imponer un estado conocido y estable para la ejecución de la aplicación. Ambas posibilidades, suponen incrementos en los tiempos de producción y elaboración, en ocasiones no considerados para la planificación o estimados de manera incorrecta, ya que la ausencia de lógica encargada de la gestión de componentes –por ejemplo, si se requiere cierto periférico para completar un servicio del sistema– no se percibe de relevancia o necesaria hasta bien avanzado el desarrollo.

Generalmente, con el objetivo de reducir los tiempos de elaboración de un producto, es frecuente reutilizar código o generar este de manera automática. Es por ello que el fabricante o el proveedor de la arquitectura, en ocasiones proporcionan un juego de herramientas o entorno de elaboración *software* con este fin y que soportan al dispositivo en cuestión. En lo que respecta a *ARM®*, este ofrece por una parte un conjunto denominado como *CMSIS* [71] (*Cortex Microcontroller Software Interface Standard*) tal que ofrecen soporte con independencia del fabricante, entre las que se incluyen herramientas tales como:

- *CMSIS-Core*, que se trata de una capa de abstracción (*Hardware Abtraction Layer* o *HAL*), tal que

habilita la reutilización de código, con objeto de reducir la curva de aprendizaje en el desarrollo de microcontroladores –en lo que respecta a particularidades que introduce el fabricante, así como aspectos estrechamente relacionados con el *hardware* y que requieren de la consulta de los respectivos manuales–, facilitando así los tiempos de comercialización para nuevos productos.

- *CMSIS-DAP*, como *firmware* para una unidad de depuración, que sirve como interfaz al puerto *DAP* (*Debug Access Port*) definido por la especificación *CoreSight Architecture* [72]

Desafortunadamente, ninguna de estas herramientas soporta la microarquitectura empleada a fecha de realización de este trabajo. Además, de los ya mencionados útiles, *ARM®* proporciona dos entornos de construción *software*, donde el primero [73] sigue un modelo de pago, mientras que el segundo [74], menos restrictivo, se conforma como un conjunto de herramientas previamente configuradas y preparadas con aquellos elementos necesarios para la construcción de aplicaciones, entre las que se encuentran *GCC* como compilador cruzado junto con sus herramientas *Binutils*, el depurador *GDB* y la implementación *Newlib* de librería estándar de *C*. Este último entorno sería aquel considerado para la elaboración de este trabajo, pero fue desechado, dado que el soporte ofrecido sólo contemplaba implementaciones para microarquitecturas*little-endian*, incompatibles con el dispositivo elegido, siendo este *big-endian*.

Reflejadas las carencias que presentan las herramientas descartadas hasta este momento, se opta, como posibles candidatas, por aquellas opciones que brinda el fabricante, de entre las cuales destaca HALCOGEN [75], cuya finalidad se remite a la generación de código de abstracción para la plataforma (*HAL*), logrando así una separación entre ámbitos de ejecución. Sin embargo, en lo relativo a la rutina de tratamiento en reinicio que proporciona esta herramienta, su utilidad es parcial, puesto que tan sólo provee de una configuración básica, sin la presencia de funciones de mayor complejidad y encargadas de comprobar el correcto funcionamiento de los propios componentes. Para suplir esta carencia, el fabricante proporciona, como solución independiente, una librería de funciones –denominada *SafeTI Hercules Diagnostic Library* [76]– encargada de aquellas características consideradas de relevancia para la seguridad y estabilidad de la plataforma –dado que en principio esta librería se diseña con ajuste a estándares como el mencionado *ISO 26262*–. La actividad de estas soluciones tiene como resultado lógica, cuyo código presenta una sintaxis específica no precisada por el estándar del lenguaje de programación, aunque si bien es cierto, esta se encuentra también soportada por otros entornos de construcción *software* ajenos al fabricante. No obstante, dicha notación exclusiva no solo afecta a aquellas partes del sistema que interactúan con la plataforma, sino que se extiende al conjunto de la aplicación, hecho que frustra uno de los objetivos planteados en este trabajo, esto es, lograr la independencia –en la medida de lo posible– de la solución frente al *hardware* subyacente y por extensión del fabricante, así como del proveedor de la arquitectura. A mayores, *Texas Intruments* añade un entorno de programación basado en *Eclipse* al que denomina *CodeComposerStudio* [77] (*CCS*), el cual agrupa bajo el mismo entorno herramientas entre las que se encuentran un compilador, un depurador y en concreto, un mecanismo para la programación de la plataforma –también provisto de forma independiente bajo la herramienta *UniFlash* [78], que escribe el *binario* elaborado en la memoria persistente del dispositivo. Este paso resulta un elemento clave e inevitable, en cuanto que la controladora de memoria es de naturaleza propietaria y por tanto opaca, de modo que resulta imposible la búsqueda de alternativas con este fin.

Establecida pues la estricta dependencia que existe al utilizar el entorno, el cual provee el fabricante, se pretende localizar una alternativa, con objeto de abordar la construcción del sistema. Al no poder prescindir del mecanismo de programación del dispositivo –aunque por suerte, no se imponen requisitos específicos para su uso–, las opciones se limitarán a las herramientas de elaboración y depuración. En lo que respecta al entorno para la construcción *software*, el candidato elegido deberá soportar el juego de instrucciones de la arquitectura empleada, así como el lenguaje de programación utilizado. Se orienta la elección a un com-

pilador cruzado –el cual se ejecuta en una plataforma distinta para la que se producen los archivos *binarios* resultantes–, siendo este una versión de *GCC*, acompañado de *Binutils* y *Newlib*, los cuales son construidos deliberadamente para el proyecto. Esta acción se lleva a cabo mediante un proceso denominado como *bootstrapping*, por el cual se genera en primera instancia un esqueleto del compilador objetivo, sin dependencias de ningún tipo, con el fin de poder compilar el resto de herramientas, entre las que se encuentra el compilador final, para el que se deberán seleccionar aquellas opciones de configuración adecuadas (*big-endian*).

Resuelta así la coyuntura que atañe a la selección del entorno para la elaboración, se propone una alternativa que afecta al proceso de depuración, siendo esta *GDB*. Se hace necesario aclarar en primer lugar que la herramienta de depuración planteada como opción, no interactúa de manera directa con el dispositivo, sino que para ello, se vale de una herramienta conocida como *OpenOCD* [79], que soporta distintos protocolos de transporte y media en la comunicación entre el entorno de desarrollo y el objetivo, el cual presenta un adaptador de depuración conocido como puerto con dicho fin, accesible mediante una interfaz y generalmente integrado. En este caso, la disposición física de la plataforma, esto es, si los puertos –en concreto la plataforma dispone de dos– para la depuración son o no accesibles, condiciona la elección aquí tomada. El primero de los puertos, CoreSight –ya mencionado con anterioridad y definido por la arquitectura–, se redirige a una interfaz propietaria denominada *XDS110* [80] incluida en la misma plataforma y contenida en otro microcontrolador, que actúa como intermediario –se observa en este caso el comportamiento de integración de sistemas embebidos en otros sistemas anfitriones– e intercepta los comandos servidos por la herramienta de depuración con el dispositivo y viceversa. Si se emplea el entorno ofrecido por el fabricante para dicha interfaz, se requiere de una mediación a través de un driver propietario, pero el resultado de la conexión es trivial, ya que son las propias herramientas las que envían de forma automática los comandos pertinentes. Sin embargo, para la propuesta *GDB*, la comunicación entre la propia herramienta y la interfaz, la lleva a cabo *OpenOCD*, de manera que este deberá conocer los detalles propietarios para el transporte y recae dicha responsabilidad en el desarrollador. Por suerte, a fecha de realización de este trabajo, existe un parche [81] que incluye esta capacidad. Sin embargo, el soporte es parcial y se pierden operaciones consideradas indispensables, como por ejemplo *cold reset*, lo que supone una limitación a la hora de exponer la información desarrollada en este trabajo. La segunda, denominada como *cJTAG*, se basa en el estándar *IEEE 1149.7* [82] y se redirige a una interfaz para su conexión, sin presencia de intermediarios, por lo que recae en el desarrollador dicha tarea, aproximación que supera con creces las motivaciones aquí presentadas y que se descarta, puesto que tampoco ha sido posible encontrar soporte no propietario. Esta situación obliga a aceptar el primer puerto como alternativa de uso junto con el depurador escogido, prescindiendo así de aquella que facilita, excepto si se alcanzan los límites de su función –puede ser necesaria la transformación de código de manera manual, dada las discrepancias existentes a nivel de sintaxis y con motivo de emplear ambos entornos–.

Comprendidas las restricciones que envuelven a la depuración del proyecto, se evalúa en este caso la cuestión referida a la portabilidad de la lógica realizada. Dado que en este trabajo se aboga por una independencia –para aquellas características que permitan una cierta autonomía frente a la plataforma– en lo que al código se refiere, hecho patente, en cuanto que se descarta en la medida de los posible el uso del entorno proporcionado por el fabricante, se han de comprender aquellas limitaciones que imponen tanto el lenguaje escogido, como el propio entorno de construcción de software. La primera se refiere al grado de portabilidad del código producido. En el lenguaje de programación *C*, se define un programa como estrictamente conforme, si y solo si utiliza las características incluidas por el estándar. Sin embargo, dichas características no actúan como restricciones de obligado cumplimiento, puesto que el propio estándar no especifica todos los resultados esperados para un comportamiento concreto del programa. Esto por ejemplo, ocasiona que

se deleguen las aproximaciones a abordar al proveedor del entorno de construcción *software* y por tanto el resultado difiere para cada uno de ellos, lo que reduce la portabilidad a dicho entorno de producción. Es posible incluso, que para diferentes iteraciones en la construcción del *software* dado el mismo entorno, se produzcan distintos resultados, lo que resulta en una inconsistencia en la elaboración –hecho que repercute por ejemplo en la extensión de las secciones de código resultantes–. En el peor de los casos, diferentes versiones del mismo entorno de construcción pueden ocasionar errores, para comportamientos validados y verificados en anteriores elaboraciones, por lo que es conveniente ajustarse a la misma versión durante toda la duración del proyecto.

Por último, la herramienta de construcción *software* considera un entorno de ejecución anfitrión en la plataforma, en el cual el inicio del programa se produce en el punto de entrada *main*, pero puesto que el sistema aquí propuesto comprende desde el inicio de la plataforma, hasta que esta delega el control a la tarea que requiere su ejecución, se produce una disonancia en la elaboración del producto, que ocasiona por ejemplo, la inserción de instrucciones al inicio de cada rutina encargadas de manejar la pila, hecho que incurre en un comportamiento indefinido, puesto que el soporte de dicha pila no se encuentra presente en los primeros compases de la ejecución, en los que se lleva a cabo el arranque del sistema. Con motivo de solventar esta casuística, el desarrollo se orienta desde una perspectiva *freestanding* para el que sólo la sintaxis del lenguaje es accesible, sin suposiciones acerca del estado del contexto. Esto, supone una solución de compromiso, dado que no se incluyen todas las características previstas en el estándar –de entre las cuales se encuentra la implementación de la librería– o lo hacen parcialmente, lo que supone un detrimento en la portabilidad, ya que el desarrollador deberá aportar una implementación propia para estas, la cual queda limitada al ámbito del propio proyecto.

6.2 Futuras líneas de trabajo

Se establece el foco en primera instancia en las aproximaciones para el desarrollo de este tipo de sistemas. Tal y como se evidencia durante la obtención de requisitos, la especificación *UML®* empleada, no permite en ocasiones la captura de información de relevancia para el ámbito de los sistemas empotrados. Con motivo de resolver esta deficiencia, la propia *OMG®* establece una extensión para *UML®* denominada *MARTE* (*Modeling and Analysis of Real-Time and Embedded Systems*) [83], tal que proporciona soporte para las fases que afectan al desarrollo aquí provisto, como obtención y especificación de requisitos, diseño y verificación/validación. *MARTE* se considera un lenguaje específico de dominio que reutiliza características particulares de*UML®* y las amplía con aspectos transversales pertenecientes a distintos niveles de abstracción, entre las que se encuentran el modelado de la plataforma subyacente –permite incorporar una representación del *hardware* al modelo–, políticas de planificación, generación de eventos por interrupción o establecimiento de precisión y dimensión de requisitos no funcionales para medidas del mundo físico entre otros. Su uso permite crear modelos extremadamente detallados, lo que facilita el proceso de transformación aquí contemplado. El uso de esta extensión para el modelado implicaría la conversión y adaptación de los diagramas propuestos –elaborados mediante *PlantUML*– de manera manual a *Papyrus*, a pesar de las limitaciones que impone en la elaboración de diagrama de secuencia –es necesario interpretar dichos diagramas o atener a las limitaciones que impone–, puesto que este sí soporta *MARTE*. Se busca, como resultado de estas acciones, lograr el soporte de transformación automática para el proyecto –hacia y desde los modelos ya existentes–, por lo que también se vuelve pertinente establecer una metodología de verificación formal –tal y como se plantea en *SeL4* [42] [84]–, a fin de comprobar el grado de adecuación de los modelos respecto de los planteamientos iniciales del problema.

Además de las unidades de protección incluidas por el proveedor de la arquitectura y estudiadas en este trabajo, los fabricantes de microcontroladores incorporan mecanismos auxiliares de protección, cuya funcionalidad a menudo coincide con aquella descrita para la *MPU*, de forma que al delegar la responsabilidad de control a dichos elementos de soporte, se liberan recursos, circunstancia que se aprovecha para la ampliación del prototipo, bajo una ambiciosa propuesta, dada la complejidad que alberga su materialización, considerada de gran interés y expuesta en lo sucesivo. Se plantea así la extensión del concepto de dominio de protección para una tarea, tal que se protegen, mediante las regiones recientemente liberadas, aquellos servicios de sistema de los que esta hace uso, de manera que estos pasan a formar parte del punto de vista actual para el proceso activo –funcionalidad similar a las *capabilities*–. Para ello, se presta necesaria la creación de una heurística, cuyo fin consiste en precisar, conocidas las tareas que hacen uso o no de manera previa un determinado servicio y las restricciones que impone la arquitectura en creación de región, la mejor posición bajo la cual albergar los servicios minimizando el número de regiones empleadas, así como su disposición. La acción de dicha heurística tendría lugar durante el proceso de elaboración del código, practicando aquellas modificaciones para el entorno de construcción *software* que sean necesarias con dicho fin.

La elaboración del prototipo sienta las bases para el análisis inicial del impacto, en términos de detrimento y rendimiento para la ejecución, dada la incorporación del mecanismo de protección. Este tipo de estudios resultan de utilidad en entornos que requieren de bajas latencias de respuesta, en los que el énfasis se establece sobre aspectos asociados a la planificación y control del tiempo de ejecución, a la par que se garantiza la estabilidad del sistema. En concreto, la ampliación del prototipo desarrollado para la arquitectura escogida, es posible mediante el uso de la denominada unidad de monitorizado de rendimiento (*PMU*), en la medida en que este mecanismo permite medir con precisión el coste en ciclos de unidad de procesamiento, en una situación de conservación de tarea e intercambio de dominios de protección, tal que participan operaciones como el acceso a memoria, la sincronización de la unidad y su posterior estabilización entre otras. El objetivo del análisis consistiría en determinar, dado una planificación estática o dinámica, qué porcentaje del tiempo total asignado para la ejecución de tarea (*quantum*) corresponde a esta sobrecarga, efecto condicionado por el tamaño de ventana de ejecución, tal que cuanto mayor es su valor, menor es el impacto, en cambio de contexto, de las operaciones asociadas al mecanismo de protección. Además, en el supuesto de planificación dinámica, se considera de interés la inclusión de una heurística capaz de asignar ventanas de protección condicionadas por el espacio restante, así como la decisión de creación y re-localización de tareas bajo demanda, para las que no se dispone de información de ejecución previo paso a su codificación.

BIBLIOGRAFÍA

[1] NXP Semiconductors: *Application Note AN12177 Power Architecture e200z4 and e200z7 Core Memory Protection Unit*, 2018.

[2] Infineon Technologies: *TriCore™ Core Architecture User Manual Volume 1*, 2012.

[3] Renesas Electronics: *Renesas RX66T Group User's Manual: Hardware*, 2019.

[4] ARM® Ltd.: *ARM® Architecture Reference Manual ARMv7-A and ARMv7-R edition*, 2014.

[5] ARM® Ltd.: *Cortex™-R5 and Cortex-R5F Revision: r1p1 Technical Reference Manual*, 2011.

[6] Leveson, Nancy G.: *Embedded System*, page 646–647. John Wiley and Sons Ltd., GBR, 2003, ISBN 0470864125.

[7] Sommerville, Ian: *Ingeniería del software*. Pearson Educación, 2005, ISBN 9788478290741.

[8] Boehm, B. W.: *A spiral model of software development and enhancement*. Computer, 21(5):61–72, 1988.

[9] Boehm, Barry: *Spiral development: Experience, principles, and refinements*. Special report, Carnegie Mellon University, July 2000. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5053.

[10] *Archived Rational Unified Process*. https://web.archive.org/web/20010605213435/http://www.rational.com/products/rup/index.jsp. Accedido: 2021-04-13.

[11] Kruchten, Philippe: *The Rational Unified Process - An Introduction, 3rd Edition*. Addison Wesley object technology series. Addison-Wesley, 2004, ISBN 978-0-321-19770-2.

[12] Boehm, B. and D. Port: *Conceptual modeling challenges for model-based architecting and software engineering (mbase)*. In *Conceptual Modeling*, 1997.

[13] Larman, Craig and Victor R. Basili: *Iterative and incremental development: A brief history*. Computer, 36(6):47–56, June 2003, ISSN 0018-9162. https://doi.org/10.1109/MC.2003.1204375.

[14] Royce, W. W.: *Managing the development of large software systems: Concepts and techniques*. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, page 328–338, Washington, DC, USA, 1987. IEEE Computer Society Press, ISBN 0897912160.

[15] *MDA® Model Driven Architecture*. https://www.omg.org/mda/. Accedido: 2021-01-28.

[16] *OMG® Object Management Group®*. https://www.omg.org. Accedido: 2021-01-28.

[17] *MOF™ Meta Object Facility*. https://www.omg.org/spec/MOF/About-MOF/. Accedido: 2021-01-28.

[18] *UML® Unified Modeling Language*. https://www.omg.org/spec/UML/About-UML. Accedido: 2021-01-28.

[19] Object Management Group®: *MDA Guide revision 2.0*, 2014.

[20] Selic, Bran: *Architectural Patterns for Real-Time Systems*, page 171–188. Kluwer Academic Publishers, USA, 2003, ISBN 1402075014.

[21] McNeile, Ashley: *Mda: The vision with the hole?* Metamaxim Ltd, 2003.

[22] OMG®: *Meta Object Facility (MOF™) 2.0 Query/View/Transformation Specification*. Specification Version 1.3, Object Management Group®, 2016.

[23] Zhang, Wei, Hong Mei, Haiyan Zhao, and Jie Yang: *Transformation from cim to pim: A feature-oriented component-based approach*. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems*, MoDELS'05, page 248–263, Berlin, Heidelberg, 2005. Springer-Verlag, ISBN 3540290109. https://doi.org/10.1007/11557432_18.

[24] Kardos, M. and M. Drozdova: *Analytical method of cim to pim transformation in model driven architecture (mda)*. Journal of information and organizational sciences, 34:89–99, 2010.

[25] Chitforoush, Fatemeh, Maryam Yazdandoost, and Raman Ramsin: *Methodology support for the model driven architecture*. In *14th Asia-Pacific Software Engineering Conference (APSEC'07)*, pages 454–461, USA, 2007. IEEE Computer Society, ISBN 0769530575.

[26] *Eclipse Papyrus™ Modeling Tool*. https://www.eclipse.org/papyrus/. Accedido: 2021-01-28.

[27] *Eclipse IDE*. https://www.eclipse.org/eclipseide/. Accedido: 2021-01-28.

[28] *PlantUML*. https://plantuml.com/. Accedido: 2021-01-28.

[29] Vardanega, T. and J. Van Katwijk: *A software process for the construction of predictable on-board embedded real-time systems*. Software: Practice & Experience, 29(3):235–266, March 1999, ISSN 0038-0644. https://doi.org/10.1002/(SICI)1097-024X(199903)29:3{%}3C235::AID-SPE231{%}3E3.0.CO;2-7.

[30] Intel Corporation: *Application Note AP-286 80186/188 Interface to Intel Microcontrollers*, 1986.

[31] Motorola Inc.: *Single-Chip Microcomputer Data*, 1984.

[32] *Processors and Microcontrollers*. https://www.nxp.com/products/processors-and-microcontrollers:MICROCONTROLLERS-AND-PROCESSORS#/. Accedido: 2021-01-29.

[33] *Microcontrollers & Microprocessors (MCUs, MPUs)*. https://www.renesas.com/us/en/products/microcontrollers-microprocessors. Accedido: 2021-01-29.

[34] *Microcontrollers & Microprocessors*. https://www.st.com/en/microcontrollers-microprocessors.html. Accedido: 2021-01-29.

[35] Douglass, Bruce Powel: *Design Patterns for Embedded Systems in C: An Embedded Software Engineering Toolkit*. Newnes, USA, 1st edition, 2010, ISBN 1856177076.

[36] Bartnes, Maria: *Safety vs. security?* In *Proceedings of the Eighth International Conference on Probabilistic Safety Assessment &Management*. ASME, 2006, ISBN 0791802442.

[37] Pan, R., G. Peach, Y. Ren, and G. Parmer: *Predictable virtualization on memory protection unit-based microcontrollers*. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 62–74, 2018.

[38] Aiken, Mark, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus: *Deconstructing process isolation*. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, MSPC '06, page 1–10. Association for Computing Machinery, 2006, ISBN 1595935789. https://doi.org/10.1145/1178597.1178599.

[39] Kumar, Ram, Akhilesh Singhania, Andrew Castner, Eddie Kohler, and Mani Srivastava: *A system for coarse grained memory protection in tiny embedded processors*. In *2007 44th ACM/IEEE Design Automation Conference*, page 218–223. IEEE, 2007, ISBN 978-1-59593-627-1. https://doi.org/10.1145/1278480.1278534.

[40] Wahbe, Robert, Steven Lucco, Thomas E. Anderson, and Susan L. Graham: *Efficient software-based fault isolation*. SIGOPS Oper. Syst. Rev., 27(5):203–216, December 1993, ISSN 0163-5980. https://doi.org/10.1145/173668.168635.

[41] Silberschatz, Abraham, Greg Gagne y Peter Baer Galvin: *Fundamentos de sistemas operativos*. McGraw Hill, 7ª edición, 2006, ISBN 84-481-4641-7.

[42] Data61: *seL4® Reference Manual Version 11.0.0*, 2019.

[43] Dhurjati, Dinakar, Sumant Kowshik, Vikram Adve, and Chris Lattner: *Memory safety without runtime checks or garbage collection*. ACM SIGPLAN Notices, 38(7):69–80, June 2003, ISSN 0362-1340. https://doi.org/10.1145/780731.780743.

[44] Maalej, Marouaand Taft, Tucker and Yannick Moy: *Safe dynamic memory management in ada and spark*. In *Reliable Software Technologies – Ada-Europe 2018*, page 37–52. Springer International Publishing, 2018, ISBN 978-3-319-92432-8.

[45] Intel Corporation: *80286 and 80287 Programmer's Reference Manual*, 1987.

[46] Intel Corporation: *80286 Hardware Reference Manual*, 1987.

[47] Shinagawa, Takahiro, Kenji Kono, and Takashi Masuda: *Fine-grained protection domain based on segmentation mechanism*. Japan Society for Software Science and Technology, 2000.

[48] Witchel, Emmett, Josh Cates, and Krste Asanović: *Mondrian memory protection*. SIGARCH Comput. Archit. News, 30(5):304–316, October 2002, ISSN 0163-5964. https://doi.org/10.1145/635506.605429.

[49] *ARM®Cortex-A series processors*. https://developer.arm.com/ip-products/processors/cortex-a. Accedido: 2021-01-18.

[50] RISC-V International: *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*, 2019.

[51] IEEE: *IEEE Standard for Information Technology – Portable Operating System Interface (POSIX™) Base Specifications, Issue 7* . Standard, Institute of Electrical and Electronics Engineers, 2018.

[52] ISO/IEC/IEEE: *Information technology — Portable Operating System Interface (POSIX™) Base Specifications, Issue 7*. Standard, International Organization for Standardization, 2009.

[53] ISO/IEC/IEEE: *Systems and software engineering – Lifecycle processes – Requirements engineering*. Standard, International Organization for Standardization, 2018.

[54] Bittner, Kurt and Ian Spence: *Use Case Modeling*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002, ISBN 0201709139.

[55] Nyßen, Alexander and Horst Lichter: *Use case modeling for embedded software systems : Deficiencies & workarounds*. In *Preliminary Proceedings of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems*, pages 63–67, January 2007.

[56] Douglass, Bruce Powel: *Real Time UML Workshop for Embedded Systems (Embedded Technology)*. Newnes, USA, 2006, ISBN 0750679069.

[57] Larman, Craig: *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, USA, 2004, ISBN 0131489062.

[58] Douglass, Bruce Powell: *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002, ISBN 0201699567.

[59] Dennis, Alan, Barbara Haley Wixom, and David Tegarden: *Systems Analysis and Design: An Object-Oriented Approach with UML*. Wiley Publishing, 5th edition, 2015, ISBN 1118804678.

[60] Rumbaugh, James, Ivar Jacobson, and Grady Booch: *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004, ISBN 0321245628.

[61] *The Linux Kernel Organization*. `https://www.kernel.org/`. Accedido: 2021-01-28.

[62] Paci, Francesco, Davide Brunelli, and Luca Benini: *Lightweight io virtualization on mpu enabled microcontrollers*. SIGBED Rev., 15(1):50–56, March 2018. `https://doi.org/10.1145/3199610.3199617`.

[63] ARM® Ltd.: *Procedure Call Standard for the Arm® Architecture – ABI 2020Q2 documentation*, 2020.

[64] Zhou, Wei, Le Guan, Peng Liu, and Yuqing Zhang: *Good motive but bad design: Why arm mpu has become an outcast in embedded systems*. ArXiv, abs/1908.03638, 2019.

[65] *The Zephyr Project™*. `https://zephyrproject.org/`. Accedido: 2021-01-28.

[66] *RTEMS Real Time Operating System (RTOS)*. `https://www.rtems.org/`. Accedido: 2021-01-28.

[67] ARM® Ltd.: *ARM® Architecture Reference Manual Supplement - ARMv8, for the ARMv8-R AArch32 architecture profile*, 2018.

[68] *TI, Texas Instruments*. `https://www.ti.com/`. Accedido: 2021-01-28.

[69] ISO: *Road vehicles – Functional safety – Parts 1–12*. Standard, International Electrotechnical Commission, 2018.

[70] IEC: *Functional safety of electrical/electronic/programmable electronic safety-related systems – parts 1–7*. Standard, International Electrotechnical Commission, 2010.

[71] *CMSIS, Cortex Microcontroller Software Interface Standard*. `https://developer.arm.com/tools-and-software/embedded/cmsis`. Accedido: 2021-01-28.

[72] *CoreSight Architecture*. `https://developer.arm.com/architectures/cpu-architecture/debug-visibility-and-trace/coresight-architecture`. Accedido: 2021-01-28.

[73] *ARM®Compiler*. `https://developer.arm.com/tools-and-software/embedded/arm-compiler`. Accedido: 2021-01-28.

[74] *GNU ARM®Embedded Toolchain*. `https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm`. Accedido: 2021-01-28.

[75] *HALCoGen*. https://www.ti.com/tool/HALCOGEN. Accedido: 2021-01-28.

[76] *SafeTI Hercules Diagnostic Library*. https://www.ti.com/tool/SAFETI_DIAG_LIB. Accedido: 2021-01-28.

[77] *Code Composer Studio*. https://www.ti.com/tool/CCSTUDIO. Accedido: 2021-01-28.

[78] *UniFlash*. https://www.ti.com/tool/UNIFLASH. Accedido: 2021-01-28.

[79] *OpenOCD, Open On-Chip Debugger*. http://openocd.org/. Accedido: 2021-01-28.

[80] *XDS110 JTAG Debug Probe*. https://www.ti.com/tool/TMDSEMU110-U. Accedido: 2021-01-28.

[81] *OpenOCD support for TI TMS570LC43x LaunchPad*. http://openocd.zylin.com/#/c/5104/. Accedido: 2021-01-28.

[82] IEEE: *IEEE Standard for Test Access Port and Boundary-Scan Architecture*. Standard, Institute of Electrical and Electronics Engineers, 2013.

[83] OMG®: *UML® Profile for MARTE™: Modeling and Analysis of Real-Time Embedded Systems*. Specification Version 1.2, Object Management Group®, 2019.

[84] *The seL4® Microkernel*. https://sel4.systems/. Accedido: 2021-01-28.

[85] *TMS570LC4357*. https://www.ti.com/product/TMS570LC4357. Accedido: 2021-01-28.

[86] *Code Composer Studio*. https://www.ti.com/tool/CCSTUDIO. Accedido: 2020-10-30.

[87] *GCC, the GNU Compiler Collection*. https://gcc.gnu.org/. Accedido: 2021-01-28.

[88] *GNU Operating System*. https://www.gnu.org/. Accedido: 2021-01-28.

[89] *GNU Binutils*. https://www.gnu.org/software/binutils/. Accedido: 2021-01-28.

[90] *Newlib*. https://sourceware.org/newlib/. Accedido: 2021-01-28.

[91] *GDB: The GNU Project Debugger*. https://www.gnu.org/software/gdb/. Accedido: 2021-01-28.

[92] *The LaTeXProject*. https://www.latex-project.org/. Accedido: 2021-01-28.

[93] *Inkscape*. https://inkscape.org/. Accedido: 2021-01-28.

[94] *Internet Archive*. https://archive.org/. Accedido: 2021-01-28.

APÉNDICE A

PLAN DE PROYECTO

Bajo este anexo se pretende dar respuesta a las necesidades propias de los primeros compases del desarrollo, en los que generalmente se demanda el plan de acción a seguir durante el proyecto. Previo a este se exige un proceso que permita el recabado de información de los denominados fundamentos teóricos, tal que sienta las bases de conocimiento para el problema tratado y habilita la elección de una metodología adecuada de desarrollo, que provee de una serie de actividades genéricas, determinando así el punto de partida para la elaboración. Una vez provistas identificadas dichas actividades, estas se descomponen en operaciones concretas y bien definidas siguiendo una aproximación *WBS*, con objeto de facilitar la obtención de sus dependencias, el orden correcto en el que se llevarán a cabo, así como los riesgos asociados a cada una de ellas –para los que además se ofrecerán un conjunto de técnicas que mitiguen o en su defecto prevengan este tipo de improvistos–. Finalmente, y tras haber completado el desglose de actividades, se propone la planificación que actuará como una hoja de ruta marcada.

A.1 MEDIOS REQUERIDOS Y ESTIMACIÓN DE COSTES

Se relatan en este apartado los recursos materiales disponibles para la elaboración del trabajo, así como algunas apreciaciones relativas al coste económico y temporal del proyecto.

A.1.1 RECURSOS HUMANOS

La disposición prevista para este trabajo contempla únicamente una sola persona para su realización, hecho que establece algunos límites para las estimaciones, sobre todo en lo que a la extensión del proyecto refiere. Además, deberá ser esta persona, por tanto el autor del mismo, la encargada de identificar y establecer la planificación para las actividades, de la elaboración de este documento, así como de la implementación del prototipo *software* considerado.

A.1.2 *Medios Hardware*

A continuación se listan algunos de los medios de apoyo, tanto *hardware* como *software*, necesarios para el desarrollo del proyecto, así como el origen para las fuentes de información empleadas:

- Ordenador portátil con las especificaciones descritas:
  - Unidad de procesamiento Intel® Core™ i7-6700HQ @ 2.60 GHz.
  - 16GB de memoria RAM.
  - Disco duro SSD de 512GB de capacidad.
- Kit de desarrollo Hercules™ TMS570LC43X *LaunchPad*:

- Microcontrolador TMS570LC4357 [85] donde los aspectos más relevantes en lo que respecta al desarrollo de este trabajo son:
  - Unidad central de procesamiento dual (ARM® Cortex®-R5F) en *lock-step* con frecuencia de operación de 300MHz.
  - 4MB de memoria persistente (*Flash*) con mecanismo de corrección de errores *ECC*.
  - 512KB de memoria principal de sistema (*SRAM*) con mecanismo de corrección de errores.
  - Ventana de espacio de 128KB para la simulación de memoria *EEPROM* con *ECC*.
  - Módulos de comunicación como *UART/SCI*, *LIN*, *SPI*, *I2C*, *DCAN* o *FlexRay*, entre otros.
  - Periféricos de sistema propietarios para el control de errores como *ESM* o *VIM*.
  - Módulo para la generación de interrupciones temporizadas de tiempo real *RTI*.
- Adaptador propietario *JTAG* [82] *XDS110* para la depuración, alojado bajo el microcontrolador Cortex®-M4.
- Potenciómetro para el control de señales bajo entradas analógicas.
- Múltiples led para la notificación visual tanto de acciones relativas a la aplicación, como de errores detectados en el sistema.

A.1.3 *Medios Software*

- Code Composer Studio [86]: se trata de un entorno de desarrollo integrado o IDE (Integrated Development Environment) ofrecido por el fabricante de la plataforma (licencia propietaria) elegida, que incluye herramientas para la compilación, depuración y librerías para las distintas interfaces *hardware*.

- *GCC (GNU Compiler Collection)* [87]: colección compuesta por compilador y las herramientas para su soporte, bajo licencia de código abierto (aunque existen versiones propietarias) y mantenido por el proyecto *GNU* [88], tal que soporta un elevado número lenguajes de programación y en concreto el lenguaje *C*, aquel escogido para esta labor. Asimismo, este *software* permite la generación de *binarios* para múltiples juegos de instrucciones y sistemas operativos.

- *GNU Binutils* [89]: conjunto de herramientas para la manipulación de formatos de archivos propios del proceso de elaboración *software* (*binarios*, código fuente o ficheros objetos, entre otros), desarrollado bajo el proyecto *GNU* y de licencia de código abierto.

- *Newlib* [90]: se trata de una implementación para la librería estándar para el lenguaje de programación *C*, cuyo uso se orienta al ámbito de los sistemas empotrados, constituida además por un compendio de licencias de *software* libre, para facilitar su integración en proyectos.

- *OpenOCD* [79]: *software* que habilita la conexión para depuración, programación o control de estado, entro otros, tal que soporta múltiples arquitecturas, así como interfaces para el enlace con los dispositivos objetivo.

- *PlantUML* [28]: herramienta bajo licencia de código abierto, que permite la creación de diversos formatos de diagramas, así como soporta múltiples estándares para su representación, a partir de texto plano, donde además permite su integración en diversos entornos de desarrollo.

- *GDB (GNU Debugger)* [91]: siendo este un depurador de licencia de código abierto para sistemas objetivos, compatible para la ejecución en sistemas *Unix-like* entre otros y desarrollado por el proyecto *GNU*, que incluye soporte para un amplio rango de arquitecturas.

- LaTeX [92]: *software* para la creación y elaboración de este documento, que emplea el uso de etiquetado y marcado para la generación de estilo y formato.
- *Inkscape* [93]: herramienta para la creación de dibujo vectorizado bajo formato *SVG*, compatible con una gran variedad de sistemas operativos y distribuida bajo licencia de código abierto.

A.1.4 Proveedores de bibliografía

- Catálogo de libros disponible mediante el sistema de préstamo de la Universidad de Valladolid, así como el servicio de préstamo digital facilitado por *Internet Archive* [94].
- Repositorios y proveedores de artículos a los que se encuentra suscrita la Universidad de Valladolid.
- Manuales de usuario y documentos técnicos de referencia que ponen a disposición los fabricantes y proveedores de arquitectura.

A.1.5 Costes

En lo que respecta a este proyecto, resulta necesaria la existencia de una plataforma *hardware* basada en microcontrolador y con soporte de unidad de protección de memoria para la ejecución del prototipo pergeñado. Es por ello que se demanda un desembolso económico con la finalidad de adquirir una placa de evaluación (en concreto el kit *Hercules TMS570LC43x*) valorada en 35 Euros para el momento de su compra. Respecto a otros posibles costes asociados, no se encuentra ningún tipo de gasto añadido en forma de salario, al tratarse este de un Trabajo de Fin de Grado, o al resto de herramientas y recursos empleados para la elaboración de este trabajo.

A.1.6 Tiempo proyectado

Dado el peso que presenta en este proyecto la extracción de información y conocimiento para el ámbito de los sistemas empotrados, así como la elaboración del prototipo que pruebe las bondades de las unidades de protección de memoria, se considera oportuno extender la duración del trabajo a un total de 500 horas, frente a las 300 establecidas de manera formal en la especificación provista para el Trabajo de Fin de Grado.

A.2 Identificación, descripción y planificación de actividades

A lo largo de este segundo apartado se pretende dilucidar el procedimiento seguido para el proyecto y de cuya aplicación resultará el sistema ofrecido como solución. En primera instancia se procederá a la introducción sumaria de las actividades contempladas, indicando qué acción se lleva a cabo para cada una de ellas, así como aquellos elementos (si los hubiera) como productos de estas. Asimismo, se dará visibilidad a los posibles riesgos que las amenazan y se proponen, siempre y cuando sea posible, procedimientos que consiguen mitigar su efecto o evitar su aparición. Por último, se incluye una planificación que involucra a todas las actividades presentadas, definiendo el tiempo previsto tomado para la realización de cada una, así como las dependencias existentes entre ellas, que delimita el flujo perseguido.

A.2.1 Actividades principales

Durante la toma de contacto inicial de un proyecto surge la necesidad de recabar información que atañe al ámbito de aplicación, con la finalidad de establecer una ontología común para el vocabulario del problema, que presenta una semántica particular, a lo largo del desarrollo y comprender aquello para lo que se pretende ofrecer solución. Es en este mismo proceso, denominado como fase propedéutica, donde se evaluarán las

distintas metodologías de desarrollo, con objeto de escoger de entre posibles candidatas aquellas acordes a las características intrínsecas y necesidades de dicho problema. Tal metodología proverá de las actividades y los productos resultantes de cada una de ellas, que permiten organizar la labor de desarrollo. En lo que atañe a este documento sólo se llevará a cabo una pequeña porción de la metodología provista, en cuanto que el resto de actividades exceden las competencias de este trabajo. Así pues, la siguiente figura presenta aquellas actividades aquí concebidas:

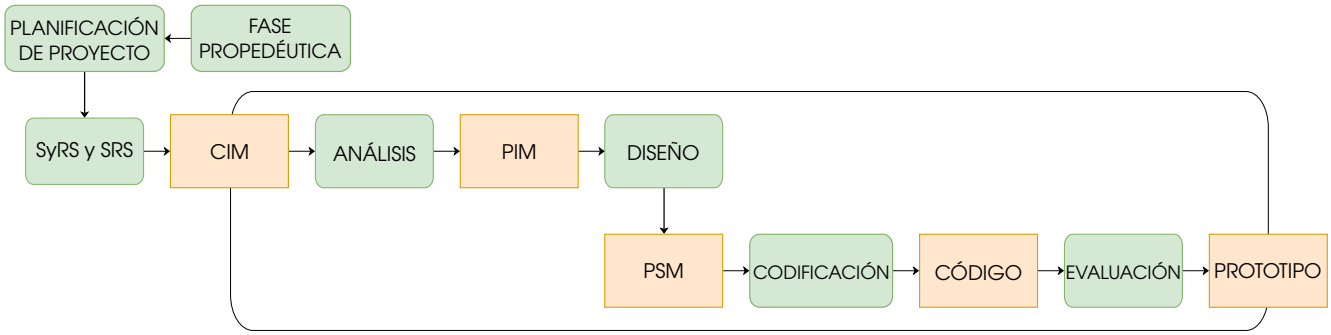

FIGURA A.1: Diagrama de actividades identificadas para el desarrollo

Como se puede comprobar, la figura A.1 se inspira en los denominados diagramas de actividad –elección debida a su semejanza–, ya que permite visualizar los pasos a efectuar (en color verde) –nodo de acción–, así como los artefactos producidos (con una tonalidad anaranjada) tras ellos –flujos de objetos–. Fuera de este diagrama quedaría la rama correspondiente al flujo principal provisto por la metodología, por lo que la representación aquí mostrada se asocia a una serie actividades que son contenidas por otra más amplia. Sin embargo, estas actividades resultan aún genéricas como para aportar valor real al desarrollo, en tanto en cuanto no es posible definir un plan de acción preciso. Dada esta deficencia se propone un proceso de descomposición jerárquico basado en *WBS* (*Work Breakdown Structure*), tal que cada nodo padre, esto es las actividades delimitadas por la metodología y referidas en la figura A.1, se divide en subactividades y estas a su vez en otras hasta encontrar aquellas pertenecientes al nivel atómico, esto es, aquellas que establecen una operación concreta. *WBS* puede orientarse a la entrega de artefactos de reducido tamaño o bien en una división que priorice los pasos mínimos. Para este caso concreto se empleará una descomposición mixta, es decir, aquella que combina ambos tipos para la elaboración de la jerarquía.

A.2.2 DESCOMPOSICIÓN EN *WBS* E IDENTIFICACIÓN DE RIESGOS ASOCIADOS

Las actividades identificadas resultantes de la aproximación *WBS* en sentido *top-down* se muestran a continuación, haciendo énfasis en el carácter jerárquico. Además, para cada una de ellas se indican posibles riesgos, junto con procedimientos que permiten mitigar o corregir su acción. Del mismo modo, se establece un rango de probabilidad de ocurrencia –siendo las posibles opciones baja, media y alta–, así como una valoración del impacto –mínimo, asumible, severo y crítico–.

A1 – Fase Propedéutica

Descripción

Fase que tras su acción asienta las bases para el desarrollo del proyecto y la manera en la que este se aborda. Permite comprender el ámbito del problema a la par que genera los fundamentos teóricos aquí recogidos.

A1.1 – Formación en fundamentos teóricos

Descripción

Esta actividad se prevé necesaria para comprender el ámbito en el que se desarrolla el problema dado, así como las restricciones que lo atañen. Asimismo, su influencia se extiende

a la elección de la arquitectura a emplear para la elaboración del prototipo. Cabe señalar que su desarrollo permitirá el establecimiento de un contexto, del cual se decide tratar el funcionamiento de los mecanismos de protección de memoria.

Artefactos

Capítulo 2.

Riesgos asociados

En este aspecto pueden diferenciarse dos riesgos que amenazan a la actividad. En concreto, el primero se refiere al espacio de tiempo dedicado a esta, donde una aproximación errónea desemboca en un pobre conocimiento acerca del ámbito de aplicación, hecho que puede derivar en una mala solución ofrecida . Mientras, el segundo se caracteriza por la imposibilidad de identificar aquellos elementos de interés dentro de los sistemas embebidos dada su extensión, lo que deriva en el primero de los riesgos asociados, al invertir un exceso de recursos en aspectos que no guardan estrecha relación con el tema a tratar.

- **Probabilidad:** Media
- **Impacto:** Crítico
- **Mitigación de impacto:** Dado que no resulta posible, dada su basta extensió, conocer de antemano la información requerida para su estudio, se proponen marcas de tiempo que delimitan la cantidad de trabajo previsto para cada área considerada de interés, de forma que puedan comunicarse los avances al tutor del trabajo para una posible reorientación.
- **Plan de actuación:** Establecer márgenes de tiempo para cada sesión, tal que permitan compensar la ausencia de extracción de conocimiento en áreas consideradas fundamentales posteriormente.

A1.2 – Elección y estudio de una metodología de desarrollo

Descripción

El empleo de procesos bien conocidos y definidos durate la elaboración de un proyecto *software* permite manejar la complejidad para el problema a resolver. Así pues, esta actividad tiene como objetivo determinar dichos procesos y conocer su aplicación.

Artefactos

NA.

Riesgos asociados

Elegir una metodología no acorde para el ámbito de aplicación y el problema dados.

- **Probabilidad:** Baja
- **Impacto:** Severo
- **Mitigación de impacto:** Cerciorarse durante la fase propedéutica de dilucidar y escoger una metodología probada para el desarollo de microcontroladores.
- **Plan de actuación:** Reutilizar aquellas actividades y artefactos compatibles tras reconsiderar la metodología.

A1.3 – Consultar proyectos existentes

Descripción

Puede ser de gran ayuda la extracción de conocimiento e información de proyectos existentes bien documentados, con objeto de identificar restricciones a las que enfrentarse durante el desarrollo o técnicas que facilitarán dicho proceso.

Artefactos

NA.

Riesgos asociados

NA.

A2 – Planificación de proyecto

Descripción

Actividad que establece la hoja de ruta seguida durante el proyecto. Descompone aquellas actividades genéricas provistas por la metodología identificada en operaciones atómicas, para las que posteriormente se elabora su planificación y se dilucidan sus riesgos.

A2.1 – Concepción inicial del proyecto

Descripción

Esta actividad permite en primera instancia definir el alcance del proyecto y sus objetivos, a fin de establecer un límite para el trabajo aquí previsto.

Artefactos

Alcance del proyecto, objetivos y motivación.

Riesgos asociados

Definir unos límites y objetivos poco realistas, ambiguos, así como subestimar las necesidades que estos requieren.

- **Probabilidad:** Media
- **Impacto:** Crítico
- **Mitigación de impacto:** Pautar con el tutor la delimitación para el proyecto y considerar las metas posibles antes de continuar con las siguientes actividades.
- **Plan de actuación:** Reajustar los límites y objetivos a unos valores con márgenes precisos y viables para el tiempo previsto en este trabajo.

A2.2 – Identificar actividades a desarrollar

Descripción

Una vez escogida la metodología para el proyecto se requiere la descomposición de los procesos genéricos en actividades concretas, las cuales podrán ser planificadas y posteriormente completadas, definiendo así una guía para el desarrollo.

Artefactos

Actividades principales.

Riesgos asociados

Considerar una estrategia de descomposición inadecuada.

- **Probabilidad:** Media
- **Impacto:** Severo
- **Mitigación de impacto:** Asegurarse de identificar y seleccionar una aproximación para el desglose acorde a las necesidades del proyecto y compatible con la metodología.
- **Plan de actuación:** Reutilizar aquellas actividades atómicas consideradas génericas respecto del ámbito de aplicación.

A2.2.1 – Descomponer actividades siguiendo *WBS*

Descripción

Donde dada la genericidad de los procesos que ofrece la metodología escogida se hace necesario dividir estos en fragmentos atómicos, con motivo de manejar la complejidad. De esta manera se escoge una aproximación de descomposición jerarquica, cuya influencia repercute en aspectos como la planificación o la identificación de riesgos.

Artefactos

Descomposición en *WBS*.

Riesgos asociados

Considerar una realización (*top-down* o *bottom-up*) incorrecta para el procedimiento dado el tipo de desarrollo.

- **Probabilidad:** Media
- **Impacto:** Severo
- **Mitigación de impacto:** Establecer una descomposición en ambas aproximaciones como primera toma de contacto, de modo que sea posible evaluar aquella que mejor se ajusta a las características del proyecto.
- **Plan de actuación:** Adaptar el flujo de actividades al tipo contrario elaborado, considerando aquellas actividades genéricas o atómicas que resulten de utilidad.

A2.2.2 – Planificar actividades mediante una representación en diagrama de *Gantt*

Descripción

Esta actividad establece el mapa de acción para el proyecto, de modo que precisa el orden bajo el que se completan las tareas y el tiempo destinado a cada una de ellas. Asimismo remarca las dependencias existentes entre ellas.

Artefactos

Diagrama de *Gantt*.

Riesgos asociados

Considerar una estimación incorrecta para cada una de las activdades identificadas.

- **Probabilidad:** Alta
- **Impacto:** Crítico
- **Mitigación de impacto:** Estudio en detalle e de manera individual para cada una de las actividades propuestas, así como las dependencias existentes entre estas.
- **Plan de actuación:** Establecer una nueva planificación teniendo en cuenta los problemas identificados y los retrasos producidos, esto es, aplicando el conocimiento adquirido.

A2.3 – Identificar riesgos asociados a las tareas propuestas

Descripción

Se pretende ofrecer posibles técnicas de mitigación y/o prevención para aquellos riesgos asociados, inevitablemente bien por su propia actividad o ajenos a esta, al propio desarrollo.

Artefactos

Compendio de riesgos para cada actividad.

Riesgos asociados

Subestimar, prescindir o descuidar la realización de la actividad.

- **Probabilidad:** Media
- **Impacto:** Crítico
- **Mitigación de impacto:** Tratar a la tarea con la debida importancia, bloqueando el resto del desarrollo hasta completar su acción si así fuese necesario.
- **Plan de actuación:** Congelar todas aquellas actividades ya iniciadas hasta proporcionar los debidos riesgos asociados.

A3 – *SyRS y SRS*

Descripción

Tarea bajo la cual se delimitan las capacidades, restricciones y servicios, que deberá ofrecer el sistema en cuestión, con objeto de satisfacer las necesidades que presentan los clientes que hacen uso de la solución.

A3.1 – Reunión para la definición y el establecimiento de las características del problema

Descripción

Esta actividad, que involucra al tutor, resulta indispensable en los primeros compases del desarrollo, en tanto en cuanto delimita la aplicación del tema a tratar, cómo será abordado y la descripción inicial del problema.

Artefactos

Descripción general del problema.

Riesgos asociados

NA.

A3.2 – Definir requisitos de sistema y *software*

Descripción

La aplicación de esta actividad consigue identificar las capacidades y limitaciones a las que deberá responder el sistema solución, cuya finalidad no es otra que la de permitir a las entidades que interactúan con este lograr sus objetivos.

Artefactos

Lista de requisitos de sistema y *software*.

Riesgos asociados

Elaborar una colección pobre de requisitos que no reflejen los requerimientos del problema en estudio.

- **Probabilidad:** Media
- **Impacto:** Crítico
- **Mitigación de impacto:** NA.
- **Plan de actuación:** Reelaborar la lista con aquellas restricciones y características identificadas en las siguientes actividades previstas para el desarrollo.

A3.3 – Especificar los casos de uso

Descripción

Esta tarea permite recoger de manera sistemática la interacción existente entre el sistema y aquellas entidades que hacen uso de este, reflejando el orden seguido, así como incorporando los requisitos previamente establecidos.

Artefactos

Requisitos de interacción. Presentación de escenarios.

Riesgos asociados

Elaborar un conjunto de casos de uso inespecíficos o que no incorporen los requisitos de interacción que demanda el proyecto.

- **Probabilidad:** Media
- **Impacto:** Crítico
- **Mitigación de impacto:** NA.
- **Plan de actuación:** Reelaborar los casos de uso incorporando tanto conocimientos previos como aquellas restricciones y características que fueron inadvertidas en el proceso.

A4 – Análisis

Descripción

Tarea agregada por diferentes actividades que permitirán comprender el ámbito de aplicación y el problema mismo. Se trata pues de un punto de inflexión, en tanto en cuanto incorpora todos los elementos generados por aquellas actividades que le preceden, sentando las bases para la concepción de la solución.

A4.1 – Identificar las entidades del problema

Descripción

Tal que su finalidad consiste en extraer la mayor información posible acerca del problema a tratar y en concreto se centra en los aspectos y elementos del negocio, los cuales resultan de interés para la solución ofrecida.

Artefactos

Modelo de dominio.

Riesgos asociados

Elaboración de un modelo de dominio que no incorpore todos los conceptos asociados al problema o que introduzca aspectos propios del dominio de la solución.

- **Probabilidad:** Media
- **Impacto:** Crítico
- **Mitigación de impacto:** NA.
- **Plan de actuación:** Considerar los participantes identificados en los casos de uso, excluyendo cualquier flujo en el que intervenga el sistema. Retomar la fase de análisis tras un nuevo estudio de la descripción general del problema, aplicando los elementos identificados y nuevos conocimientos adquiridos.

A4.2 – Establecer el flujo general del sistema

Descripción

Actividad cuyo desarrollo proporciona un punto de partida, de modo que identifica el comportamiento general de la aplicación, incorporando la información provista previamente en los casos de uso, pero bajo una perspectiva estructural.

Artefactos

Flujo principal del sistema.

Riesgos asociados

No considerar e incorporar todos los flujos identificados bajo los casos de uso.

- **Probabilidad:** Media
- **Impacto:** Aceptable
- **Mitigación de impacto:** Desarrollar diagramas de secuencia que incluyan la descripción gráfica de los escenarios, previo paso a la concepción de este.
- **Plan de actuación:** Cerciorarse de incluir todos los flujos y funcionalidades dispuestas durante la extracción de requisitos.

A4.3 – Definir heurísticas

Descripción

Esta actividad permite delimitar técnicas para el tratamiento de aspectos concretos del problema, que requieren conocimientos de disciplinas transversales.

Artefactos

Heurística de planificación

Riesgos asociados

NA.

A5 – Diseño

Descripción

Siendo este proceso aquel bajo el que se agrega toda la información relativa a las tecnologías empleadas y que permitirán implementar la solución concebida. Así pues, se consigue adaptar los artefactos resultantes de anteriores procesos a la plataforma y paradigma de programación escogidos.

A5.1 – Concebir la distribución arquitectónica de la aplicación

Descripción

En concreto esta tarea permite agrupar de manera lógica en base a criterios como la cohesión, esto es, aquellas con comportamientos similares o el acomplamiento, donde en concreto se pretende limitar el conocimiento que tienen las distintas partes de la solución del resto de componentes.

Artefactos

Diagrama de arquitectura.

Riesgos asociados

Identificar y aplicar un patrón de arquitectura que dificulte la extensibilidad y escalabilidad del sistema.

- **Probabilidad:** Media
- **Impacto:** Aceptable
- **Mitigación de impacto:** Considerar sólo como opciones aquellos patrones arquitectónicos probados en el ámbito de los sistemas empotrados.
- **Plan de actuación:** Rediseñar la arquitectura previamente efectuada incluyendo siempre que sea posible aquellos aspectos compatibles y que puedan ser de utilidad.

A5.2 – Realizar el diagrama de clases en representación de los componentes de la aplicación

Descripción

Actividad que pretende identificar los distintos componentes que conforman el producto, así como reflejar sus atributos y operaciones, junto a las relaciones existentes, de modo que se visibiliza la colaboración entre ellos.

Artefactos

Representación de los componentes del sistema.

Riesgos asociados

Utilizar patrones de diseño que no se ajusten a las características de la solución propuesta y para el ámbito de aplicación.

- **Probabilidad:** Media
- **Impacto:** Severo
- **Mitigación de impacto:** Considerar sólo como opciones aquellos patrones probados en el ámbito de los sistemas empotrados.
- **Plan de actuación:** Rediseñar el modelo incorporando aquellos componentes identificados y que permitan su reutilización con independencia de la aproximación tomada.

A6 – Codificación

Descripción

Tarea que tiene por objeto la realización del prototipo como propuesta para este proyecto.

A6.1 – Procurar las herramientas de programación

Descripción

Previo paso a la elaboración de la solución *software* se necesitarán las correspondientes herramientas que cubran todas las fases de la implementación. En concreto, la búsqueda de posibles candidatos se apoya en aspectos como la portabilidad, compatibilidad y tipos de licencia, entre otros.

Artefactos

Compilador, depurador y programadora del *firmware*.

Riesgos asociados

Elegir herramientas con características de pago y licencias restrictivas, así como aquellas que ofrezcan únicamente un soporte parcial para la plataforma escogida.

- **Probabilidad:** Media
- **Impacto:** Severo
- **Mitigación de impacto:** Seleccionar herramientas de desarrollo *software* con licencias permisivas
- **Plan de actuación:** Adaptar aquella codificación realizada bajo este tipo de herramientas a la sintaxis de aquellas más persivas y recompilar de nuevo. Si fuese necesario, deberán construirse las herramientas de manera que permitan el soporte de la plataforma.

A6.2 – Integrar bibliotecas requeridas y compatibles para la plataforma

Descripción

Además de las herramientas provistas para la labor de desarrollo, es posible el empleo de bibliotecas de código, las cuales recogen funcionalidad reutilizable propia del ámbito de los sitemas embebidos, hecho que permite reducir en gran medida los tiempos de desarrollo, así como evita la introducción de errores durante este proceso.

Artefactos

Biblioteca estándar de *C*.

Riesgos asociados

Utilizar bibliotecas con licencias propietarias o de elevado tamaño para los requerimientos del binario esperado.

- **Probabilidad:** Media
- **Impacto:** Severo
- **Mitigación de impacto:** Realizar un estudio previo que determine el tipo de licencia empleada y comprobar si es posible su edición o en su defecto existen opciones de compilación que reducen la extensión del binario.
- **Plan de actuación:** Emplear bibliotecas de desarrollo o sustituir dicha funcionalidad por código propio. Sustituir aquellas bibliotecas por soluciones menos restrictivas.

A6.3 – Desarrollar el prototipo

Descripción

Tarea asociada a la implementación del producto ofrecido como solución.

Artefactos

Prototipo de prueba.

Riesgos asociados

NA.

A6.3.1 – Definir el *bootloader*

Descripción

De manera previa a la existencia de cualquier tipo de entorno de ejecución y para el alojamiento de las tareas que tienen cabida se demanda un estado de conocido y confiable para la plataforma, labor que llevará a cabo el *bootloader*.

Artefactos

Bootloader.

Riesgos asociados

NA.

A6.3.2 – Construir el *kernel* mínimo

Descripción

Una vez definido el estado de ejecución inicial se procede a la elaboración del entorno que ofrecerá los servicios requeridos por las entidades externas. En este caso concreto no sólo conlleva el soporte a la ejecución de tareas, sino también a sus dominios de protección.

Artefactos

Kernel básico.

Riesgos asociados

NA.

A6.3.3 – Implementar las tareas que simulan los casos de estudio

Descripción

El desarrollo de esta actividad permite elaborar un mecanismo capaz de visibilizar las bondades y capacidades del sistema provisto para aquellos aspectos que competen a las unidades de protección de memoria.

Artefactos

Aplicación de usuario.

Riesgos asociados

NA.

A7 – Evaluación

Descripción

Provisto el código que constituye el prototipo, resulta indispensable analizar y confirmar el grado de adecuación respecto de los requisitos establecidos. Del mismo modo, esta tarea permite identificar posibles comportamientos no previstos o anómalos, para su posterior mitigación o en prevención en el mejor de los casos.

A7.1 – Desplegar el programa en la plataforma

Descripción

Esta actividad resulta indispensable para la comprobación del prototipo desarrollado, puesto que en última instancia el código de la aplicación deberá residir en la plataforma, con la finalidad de producir valor.

Artefactos

NA.

Riesgos asociados

NA.

A7.1.1 – Programar la memoria persistente con el *firmware* resultante

Descripción

Este es el primer paso a la hora de proseguir con cualquier actividad de análisis del producto. Dado que al retirar la energía del dispositivo este pierde el contenido de la memoria principal se recurre, en este caso, a la *flash*, de modo que pueda estudiarse hasta el proceso de arranque mismo, así como la labor de protección.

Artefactos

NA.

Riesgos asociados

NA.

A7.1.2 – Establecer comunicación con el mecanismo de depuración

Descripción

Provisto el código que deberá albergar la plataforma se procede a la conexión para la depuración, que permitirá controlar la ejecución y facilitará la labor de análisis contemplada.

Artefactos

Canal de depuración.

Riesgos asociados

NA.

A7.2 – Ejecutar el *software*

Descripción

Con motivo de permitir el posterior análisis y evaluación se dispone esta operación en forma de pasos secuenciales, de modo que la ejecución por completo resulta controlada manualmente a voluntad, haciendo hincapié en aquellas secciones consideradas críticas.

Artefactos

NA.

Riesgos asociados

NA.

A7.3 – Analizar el resultado de la ejecución

Descripción

Actividad que junto con la anterior permite comprender el estado de la plataforma y los efectos para cada uno de los pasos comprendidos entre secciones críticas.

Artefactos

Lista de errores y fallas en la ejecución.

Riesgos asociados

No se dispone de una metodología de pruebas y análisis correcta o eficaz, tal que permita detectar posibles condiciones anormales y no previstas.

- **Probabilidad:** Media
- **Impacto:** Severo
- **Mitigación de impacto:** Establecer un plan de pruebas previo paso al análisis, de forma que se delimita en primera instancia cada paso de la ejecución y los eventos esperados.
- **Plan de actuación:** Congelar el análisis hasta que se dispone de un plan de pruebas completo.

A.2.3 Planificación prevista

Las figuras A.2 y A.3 caracterizan la planificación prevista para el trabajo a partir de las actividades previamente identificadas. Asimismo, se muestra tanto la relación de precedencia como la jerarquía existente de manera gráfica existente entre las distintas tareas, donde para cada se indican las fechas de inicio y fin esperadas.

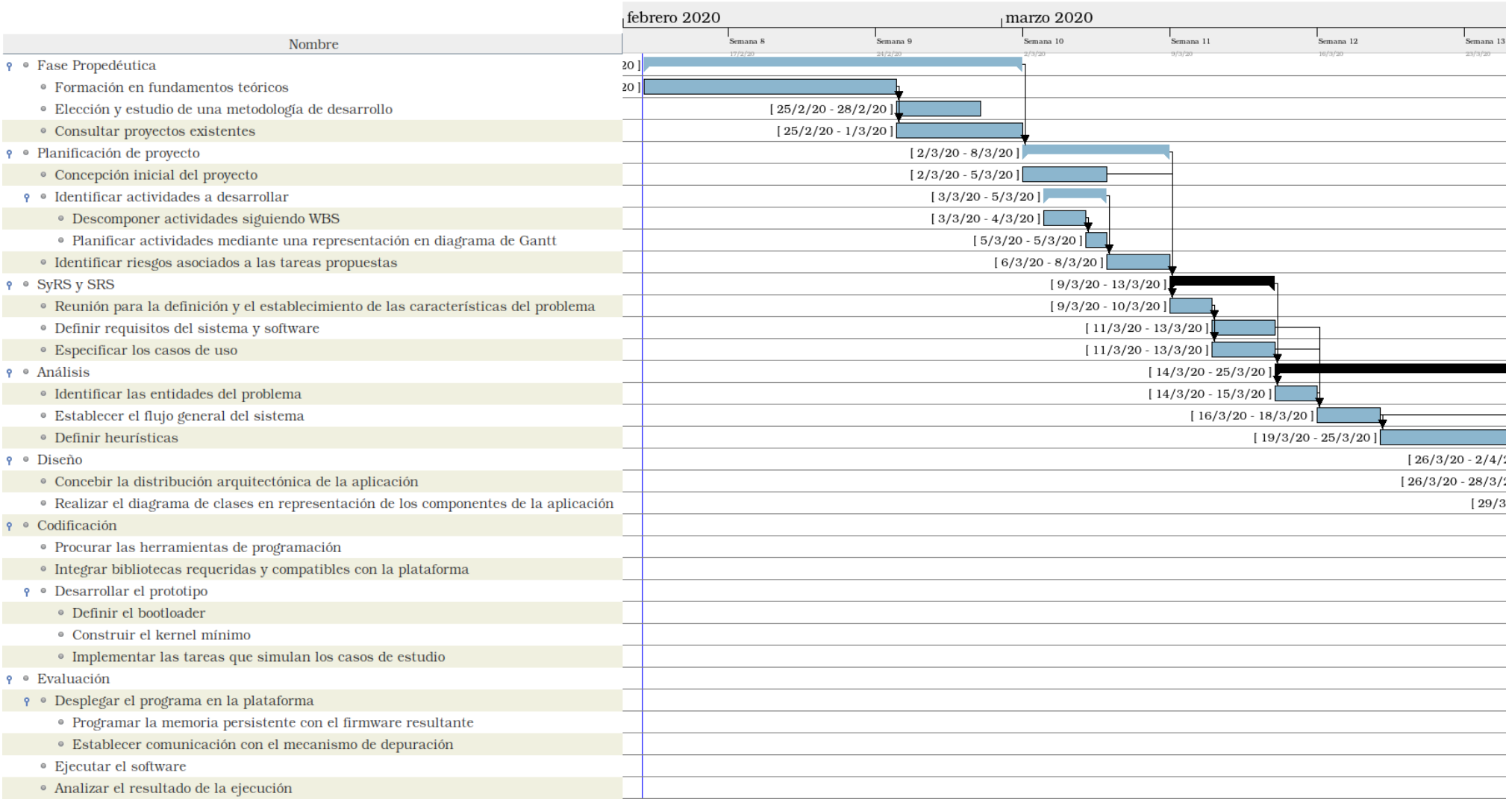

FIGURA A.2: Planificación prevista representada mediante diagrama de Gantt 1/2

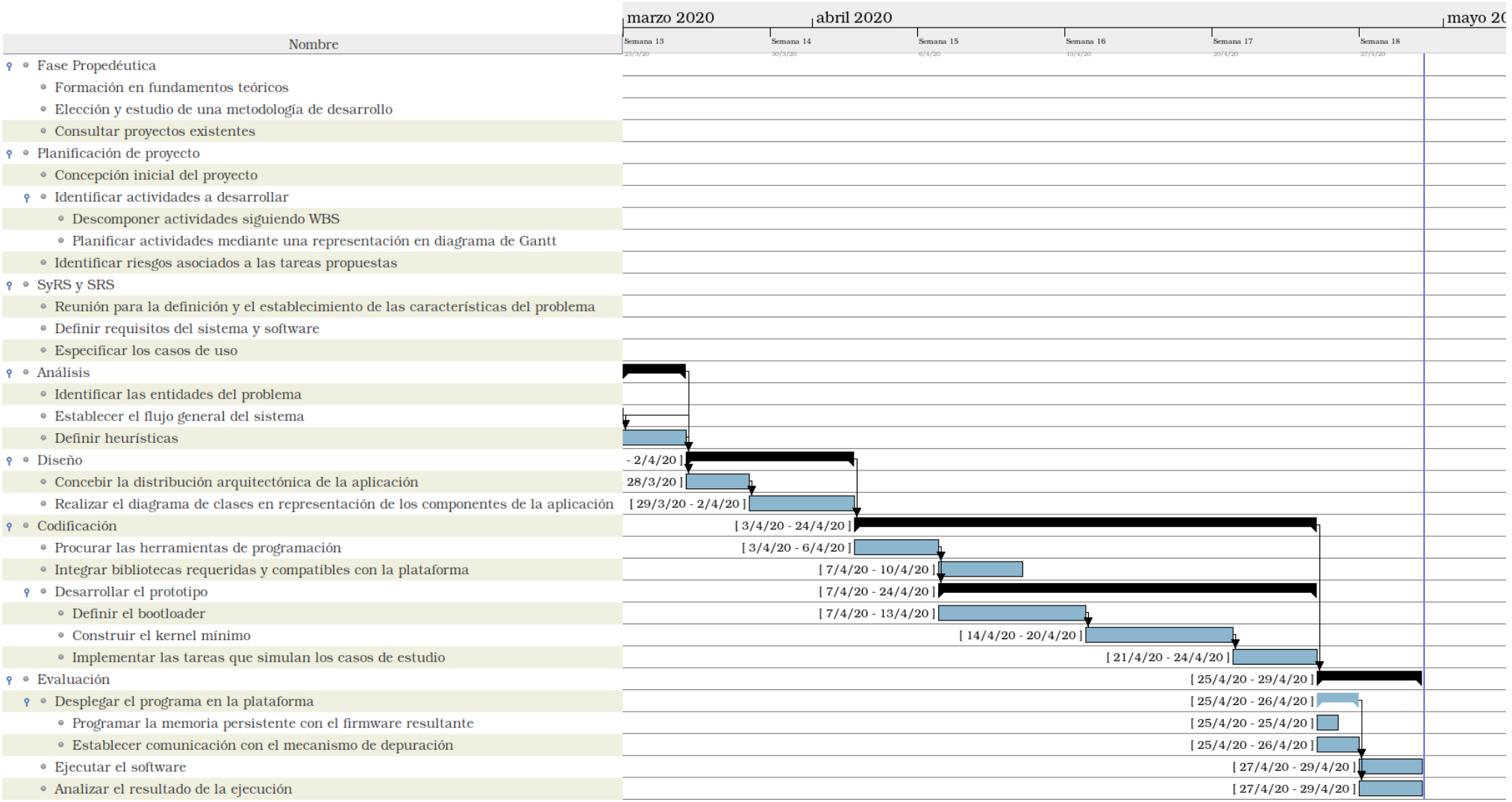

FIGURA A.3: Planificación prevista representada mediante diagrama de Gantt 2/2