

eman ta zabal zazu



Universidad del País Vasco Euskal Herriko Unibertsitatea

Departamento Ingeniería de Sistemas y Automática

Escuela de Ingeniería de Bilbao

MAS-RECON: PLATAFORMA DE GESTIÓN DE APLICACIONES DISTRIBUIDAS, ADAPTABLES Y SENSIBLES AL CONTEXTO

TESIS DOCTORAL

D. Unai Gangoiti Gurtubay

Directoras: Marga Marcos Muñoz

Aintzane Armentia Díaz de Tuesta

Bilbao, 2022

En primer lugar, quiero expresar total gratitud a mis directoras de tesis Marga y Aintzane por su paciencia, esfuerzo, apoyo y dedicación. Su confianza y ayuda, indispensable en las últimas fases, es lo que ha hecho posible llevar a término este trabajo. Además, sus conocimientos en el campo de aplicación han sido imprescindibles para hacer posible lo que en algunos momentos parecía imposible.

Por otra parte, agradecimiento especial a los compañeros de Departamento de Ingeniería de Sistemas y Automática: a todos los profesores, doctorandos y ya doctores con los que he compartido muy de cerca largas jornadas en el laboratorio de Control 1.

En general, quiero agradecer su apoyo a todas y cada una de las personas que han vivido y compartido conmigo la realización de este trabajo.

RESUMEN

En esta Tesis Doctoral se presenta MAS-RECON, una plataforma *genérica, personalizable y extensible* para la gestión del ciclo de ejecución de aplicaciones sensibles al contexto, que incluye desde el registro de sus elementos hasta su puesta en marcha y la gestión de su ejecución. Estas aplicaciones se encargan de la supervisión de su entorno para detectar cambios en él y poder responder de forma rápida y adecuada.

En estas aplicaciones, habitualmente los datos son adquiridos por dispositivos ligeros ubicados en el entorno a supervisar, mientras que su procesamiento suele realizarse mediante equipos más potentes y alejados. Para asegurar la fiabilidad de la información obtenida, tanto la adquisición de estos datos como su procesamiento y transmisión pueden estar sometidos a ciertos requisitos temporales y de seguridad, siendo necesario mantener la disponibilidad del servicio, incluso cuando se producen fallos de hardware. Además, se trata de aplicaciones dinámicas cuyo número y tamaño puede evolucionar con el tiempo, para poder adaptarse a las circunstancias del contexto en cada instante. Incluso, cuando se produce un cambio relevante, deben poder responder no sólo modificando su propia ejecución, sino también actuando sobre la ejecución de otras aplicaciones. Sin embargo, además de estos requisitos comunes, al ser aplicaciones que pueden estar presentes en ámbitos de aplicación muy diversos (desde la monitorización remota de la salud o la detección temprana de catástrofes, hasta sistemas de video-vigilancia o sistemas de fabricación flexibles), también demandan requisitos específicos de cada ámbito.

En la literatura se pueden encontrar propuestas de plataformas de gestión que cubren algunas de estas demandas, pero no todas. Y, además, para hacer frente a las necesidades específicas del dominio se suelen desarrollar soluciones ad-hoc que no se pueden generalizar a otros ámbitos. En este contexto, MAS-RECON dispone de un *núcleo genérico* que cubre las *necesidades operacionales* (arranque, parada, operación normal y gestión del estado del sistema) y de *flexibilidad* (auto-adaptabilidad

conducida por la propia aplicación y disponibilidad en base a réplicas) de las aplicaciones sensibles al contexto de cualquier campo. MAS-RECON también contempla la *variabilidad del dominio*, ya que se ha diseñado para que pueda ser personalizable e incluso extensible a dominio.

Para todo ello, MAS-RECON se basa en dos tecnologías. La *tecnología multi-agente* permite la implementación de inteligencia distribuida con supervisión centralizada. De esta forma se puede distribuir la toma de decisiones a entidades del dominio al mismo tiempo que se asegura un correcto estado global del sistema. De hecho, MAS-RECON da un paso más que otras plataformas de la literatura, centrando su gestión de la ejecución en las propias aplicaciones y no en sus componentes. Esto es posible gracias a la *tecnología de meta-modelado* que posibilita la inclusión del concepto de aplicación en los mecanismos de gestión. De hecho, a pesar de que el concepto de aplicación es específico de cada ámbito, MAS-RECON proporciona la definición y gestión genérica del estado global del sistema gracias a la definición de meta-modelos de dominio. Estos meta-modelos tienen que seguir la estructura establecida por MAS-RECON, siendo las entidades que los componen diferentes en cada ámbito.

En resumen, con el objetivo de desarrollar plataformas de gestión propias de dominio, MAS-RECON propone: 1) núcleo de arquitectura genérico, donde se distinguen agentes de dominio dotados de inteligencia y agentes de sistema para su supervisión; 2) esqueleto del código de los agentes de dominio así como el API de los agentes de supervisión; y 3) metodología de personalización y extensión a dominio, basada en la definición del meta-modelo de dominio y el desarrollo de plantillas de agentes a partir del código proporcionado.

ABSTRACT

This research work presents MAS-RECON, a *generic, customizable, and extensible* platform for managing the executions cycle of context-aware applications, which includes everything from the registration of its elements to its start-up and execution management. These applications are responsible for monitoring the environment to detect changes and respond quickly and appropriately.

In these applications, data is usually acquired by light devices located in the environment to be supervised, meanwhile the processing is usually performed by powerful and remote equipment. To ensure the reliability of the obtained information, both the acquisition of this data and its processing and transmission may accomplish certain time and security requirements, making it necessary to maintain the availability of the service, even when hardware failures occur. In addition, to be able to adapt to any circumstance applications are dynamic, and their number and size can evolve over time. Even when a relevant change occurs, they must be able to respond not only by modifying their own execution, but also by acting on the execution of any other application. However, in addition to these common requirements, and considering that applications can be used in diverse domains (from remote health monitoring or early detection of catastrophes to video-surveillance systems or flexible manufacturing systems), they also demand domain specific requirements.

Proposals for management platforms that cover some of these demands, but not all of them, can be found in the literature. And furthermore, to address the specific needs of the domain, and-hoc solutions that cannot be generalized to other areas are usually developed. In this context, MAS-RECON has a *generic core* that covers *operational needs* (start-up, shutdown, normal operation, and system state management) and *flexibility* (self-adaptation driven by the application itself and availability based on replicas) of context-sensitive applications in any domain. MAS-RECON also supports the *variability of the domain* since it has been designed to be customizable and even extensible to domain.

For all this, MAS-RECON is based on two technologies. *Multi-agent technology* allows the implementation of distributed intelligence with centralized supervision. In this way, decision-making can be distributed to domain entities while ensuring a correct global state of the system. In fact, MAS-RECON goes one step further than other platforms in the literature, focusing its execution management on the application themselves and not on their components. This is possible thanks to the *meta-modeling technology* that makes it possible to include the application concept in the management mechanisms. Fairly, even though the application concept is specific to each domain, MAS-RECON provides the generic definition and management of the global state of the system by means of the definition of domain meta-models. These meta-models must follow the structure established by MAS-RECON, being the entities that compose them different in each area.

In summary, with the aim of developing domain specific management platforms, MAS-RECON proposes: 1) generic architecture core, where domain agents with intelligence and system agents for their supervision are distinguished; 2) skeleton of the code of the domain agents as well as the API of the supervision agents; 3) customization and domain extension methodology, based on the definition of the domain meta-model and the development of agent templates based on the provided code.

ÍNDICE

ÍNDICE DE CONTENIDOS

1	INTRODUCCIÓN	
1.1	MOTIVACIÓN	1-1
1.2	ESTADO DEL ARTE	1-5
1.2.1	<i>Requisitos Operacionales (R1-R3)</i>	<i>1-7</i>
1.2.2	<i>Requisitos No-Operacionales.....</i>	<i>1-10</i>
1.3	OBJETIVOS.....	1-18
1.4	RESULTADOS Y DISCUSIÓN	1-19
1.4.1	<i>Inteligencia Distribuida: Requisitos Operacionales (R1-R3).....</i>	<i>1-21</i>
1.4.2	<i>Supervisión Centralizada: Trazabilidad (R6) y Variabilidad de Dominio (R8) 1-25</i>	
1.4.3	<i>Flexibilidad: Auto-adaptabilidad (R5) y Auto-recuperación (R7).....</i>	<i>1-32</i>
1.4.4	<i>Personalización y Extensión</i>	<i>1-37</i>
1.4.5	<i>Análisis de Rendimiento</i>	<i>1-41</i>
1.5	REFERENCIAS.....	1-45
2	CONCLUSIONES	
2.1	CONCLUSIONES	2-1
2.2	TRABAJO FUTURO.....	2-3
3	ANEXO: PUBLICACIONES	
3.1	FLEXIBILITY SUPPORT FOR HOMECARE APPLICATIONS BASED ON MODELS AND MULTI-AGENT TECHNOLOGY	3-1
3.2	MODEL-DRIVEN DESIGN AND DEVELOPMENT OF FLEXIBLE AUTOMATED PRODUCTION CONTROL CONFIGURATIONS FOR INDUSTRY 4.0.....	3-29
3.3	A CUSTOMIZABLE ARCHITECTURE FOR APPLICATION-CENTRIC MANAGEMENT OF CONTEXT-AWARE APPLICATIONS.....	3-59

ÍNDICE DE FIGURAS

Figura 1: Arquitectura de plataforma propuesta para la gestión de sistemas de monitorización remota de la salud (eHC) en P.1 y empleada para la gestión de sistemas de video-vigilancia en P.4.....	1-23
Figura 2: Arquitectura de plataforma propuesta para la gestión de sistemas de fabricación flexible (FMS), en P.2 y P.4.....	1-23
Figura 3: Arquitectura de la plataforma MAS-RECON: genérica, personalizable y extensible para la gestión de aplicaciones sensibles al contexto	1-24
Figura 4: Estructura de repositorio para el estado global del sistema propuesta en P.1 para sistemas eHC.....	1-26
Figura 5: Estructuras de repositorio para el estado global del sistema propuesta en P.3 para sistemas multi-media.....	1-26
Figura 6: Estructuras de repositorio para el estado global del sistema propuesta en P.2 para sistemas FMS.....	1-27
Figura 7: Meta-modelo que define la estructura genérica del repositorio de sistema (SR).	1-28
Figura 8: API genérica del <i>System Repository Agent</i>	1-29
Figura 9: Diagrama de secuencia del arranque de entidades de recurso y de aplicación.....	1-30
Figura 10: Interfaces definidos para las interacciones entre Agentes de Supervisión de Sistema (en color azul) y Agentes de dominio de Aplicación y Recurso (en color verde), en relación con los requisitos de flexibilidad: auto-adaptabilidad (R5) y auto-recuperación (R7).	1-34
Figura 11: Diagrama de secuencia de los mensajes intercambiados durante el proceso de auto-recuperación: desde la detección del fallo hasta su recuperación.	1-37
Figura 12: Diagramas de estados (FSM) correspondiente al comportamiento genérico de los agentes de domino: a) de recurso y b) de aplicación.	1-39

Figura 13: Tiempos de planificación (a), despliegue (b) y arranque (c) de componentes en MAS-RECON (color azul), en comparación con la plataforma de orquestación Kubernetes (color naranja), en función de la carga de trabajo del sistema. 1-42

Figura 14: Tiempos de planificación (a), despliegue (b) y arranque (c) de componentes en MAS-RECON (color azul), en comparación con la plataforma de orquestación Kubernetes (color naranja), en función del número de nodos del sistema. 1-43

ÍNDICE DE TABLAS

Tabla 1: Requisitos de la plataforma.....	1-6
Tabla 2: Cumplimiento de requisitos por las plataformas de gestión de aplicación...1-9	
Tabla 3: Relación objetivos - publicaciones.....	1-20
Tabla 4: Tiempos de recuperación ante fallos de aplicaciones de diferente tamaño (N). El fallo únicamente afecta a uno de los nodos (i.e., pérdida de N/6 componentes)	1-45

1 INTRODUCCIÓN

1.1 Motivación

Los actuales avances en las tecnologías de la información y comunicación han permitido la expansión del llamado Internet de las Cosas (*Internet of Things*, IoT) (Čolaković and Hadžialić, 2018; Perera et al., 2014) y su variante industrial IIoT (*Industrial IoT*) (Boyes et al., 2018; Xu et al., 2018). Este paradigma está basado en la interconexión universal de “objetos” o “cosas” dotadas de identidad digital, y con la habilidad de medir, comprender, procesar y reaccionar a su entorno, pudiendo también colaborar para lograr objetivos comunes. IoT e IIoT han permitido, por lo tanto, el desarrollo de aplicaciones sensibles al contexto que pertenecen a dominios de aplicación muy diferentes, que van desde la monitorización remota en la prevención de desastres naturales (Ray et al., 2017; Shah et al., 2019) o la supervisión médica (Baker et al., 2017; Islam et al., 2015), hasta la agricultura inteligente (Ayaz et al., 2019; Farooq et al., 2019) o los sistemas de fabricación flexible (*Flexible Manufacturing Systems*, FMS) (Chen et al., 2018; Qi and Tao, 2019).

Estas aplicaciones, a pesar de encontrarse en ámbitos de aplicación tan dispares, presentan requisitos comunes. Por un lado, los datos de contexto generalmente son registrados por dispositivos integrados, como dispositivos IoT e IIoT, que se ubican generalmente cerca del entorno físico bajo supervisión. Evidentemente, lo que puede variar son los dispositivos utilizados para la toma de datos de contexto, aunque todos aportan el mismo tipo de funcionalidad. Por otro lado, las tareas de procesamiento de dichos datos pueden requerir para su ejecución equipos de medio-alto rendimiento, físicamente alejados de las medidas (*distribución y heterogeneidad de nodos*). Por ejemplo, en la predicción de erupciones volcánicas, los sensores se colocan en el cráter, pero la información capturada se analiza en una ubicación remota utilizando, por ejemplo, algoritmos de “machine learning”. Así mismo, algunos sistemas de riego disponen de sensores inalámbricos para monitorizar la humedad del suelo y del aire, y drones con cámaras. Esta información se analiza en un centro de procesamiento dentro de la finca, con el fin de tomar decisiones sobre cambios de riego, evitando el

desperdicio de agua y mejorando la calidad y cantidad de cultivos. Por lo tanto, de forma general, las aplicaciones estarán formadas por un conjunto de componentes software desplegados en nodos diferentes que tienen que comunicarse y cooperar, a veces con limitaciones de tiempo (*requisitos de tiempo*). Además, es posible que estas aplicaciones deban evolucionar con los cambios de contexto para adaptarse a nuevas situaciones (*adaptabilidad*). Por ejemplo, en los sistemas de alerta temprana se puede adaptar la resolución temporal, espacial y numérica de los sensores activos al nivel de criticidad. Así, será posible procesar la información con mayor detalle cuando se detecte una situación de peligro. En el mismo sentido, en caso de una alarma de incendio en una residencia de mayores, sería interesante activar la monitorización remota de constantes vitales de personas mayores, dotando así a los equipos de emergencia de información útil para la gestión de la crisis. Esta adaptación implica cambios precisos en la funcionalidad de la aplicación.

A veces, incluso es necesaria la cooperación entre diferentes aplicaciones para monitorizar correctamente el entorno y/o reaccionar a los cambios que se detecten en él. Se trata de sistemas cambiantes en el sentido de que las aplicaciones y los recursos pueden incorporarse o apagarse con el tiempo (*escalabilidad*); cambiando, en consecuencia, la demanda y/o la disponibilidad de recursos. Como ejemplo, en el caso de sistemas de alerta temprana, un aumento en el nivel de criticidad puede implicar la activación dinámica de más sensores. En el caso de residencias de mayores, será necesario ajustar recursos a medida que el número de personas varíe o se requieran nuevas tareas de seguimiento debidas a la evolución del estado de salud de las personas.

Además, debido a la naturaleza sensible de la información capturada y procesada, resulta esencial el garantizar la seguridad y privacidad de los datos (*seguridad*). Por ejemplo, para que los datos médicos solo puedan ser accesibles por el personal autorizado. O en las fábricas inteligentes, donde la falta de seguridad puede provocar pérdidas económicas, acceso no autorizado a datos confidenciales o incluso lesiones y la muerte de personas.

Finalmente, es fundamental mantener la *disponibilidad* del servicio, minimizando las interrupciones y recuperando la ejecución de la aplicación desde el punto de parada, incluso en caso de fallos de hardware. En este sentido, en sistemas de fabricación flexible, cuando falla una máquina y para reducir las pérdidas económicas, se debe reprogramar el plan de fabricación lo antes posible. Lo mismo ocurre en la supervisión de salud crítica, si se quiere evitar una situación de peligro para el paciente.

Además de estos requisitos comunes, las aplicaciones objeto de este trabajo también presentan particularidades de su propio dominio, empezando por el propio concepto de aplicación. Por ejemplo, en los sistemas de salud, una aplicación comprende el conjunto de las tareas de seguimiento médico necesarias para supervisar la salud de una persona. Sin embargo, en sistemas de fabricación, una aplicación puede entenderse como las tareas de seguimiento de la fabricación de un conjunto de productos. En ambos dominios, además, puede ser necesario resolver eventos sobrevenidos, como la detección de un problema de salud o un fallo en un recurso de fabricación. Aunque el objetivo en ambos casos es recuperar una situación normal, la forma de alcanzarla y las entidades de dominio implicadas pueden ser muy diferentes.

La estructura de la aplicación también depende del dominio, ya que las aplicaciones se componen de un conjunto de entidades que colaboran para lograr los objetivos funcionales de la aplicación. Y, dependiendo del dominio, las entidades pueden tener una relación jerárquica determinada. Adicionalmente, las aplicaciones pueden tener requisitos no-funcionales que afectan a todas sus entidades. A modo de ilustración, en un sistema de detección de incendios, si se detecta que la temperatura de una zona aumenta, se puede incrementar el número de sensores en funcionamiento y realizar un procesamiento más complejo de la información capturada para confirmar la existencia de fuego. Esto implica que los requisitos de temporización (cada cuánto tomar muestra) y de procesamiento de la aplicación cambien con el nivel de criticidad.

Los recursos hardware necesarios también pueden depender del dominio de aplicación. En el caso de la monitorización sanitaria, ésta puede ser realizada por

equipos con mayor o menor capacidad de procesamiento, algunos de los cuales deben estar conectados a hardware específico como sensores o cámaras. Mientras que, en los sistemas de fabricación, los recursos necesarios suelen ser las propias máquinas de fabricación, robots de ensamblaje, las fresadoras/taladradoras, transportes, etc.

Desde el punto de vista de la implementación, ya desde primeros de siglo se pueden encontrar trabajos que usan diferentes paradigmas de arquitecturas de software distribuidas para desarrollar aplicaciones sensibles al contexto. Así, por ejemplo, se han utilizado sistemas basados en componentes (Vale et al., 2016), sistemas multi-agente (*Multi-Agent System, MAS*) (Michael Wooldridge, 2009), arquitecturas orientadas a servicios o microservicios (James Lewis and Martin Fowler, 2014). Cualquier implementación de una arquitectura de software distribuida cumple con los requisitos de distribución, heterogeneidad, escalabilidad y requisitos temporales, y puede ser extendida con prestaciones de seguridad. También da soporte al inicio, parada y comunicación entre sus módulos distribuidos. Para hacer frente a los requisitos de adaptabilidad y disponibilidad se pueden añadir mecanismos de reconfiguración a plataformas existentes. La reconfiguración dinámica permite modificar la configuración de la aplicación cuando ocurre una situación anormal, como un cambio de contexto o un fallo hardware. Sin embargo, según la literatura, ninguna plataforma construida sobre estas arquitecturas de software distribuidas cubre todos los requisitos identificados para las aplicaciones sensibles al contexto. Específicamente, no se ha encontrado ninguna en la literatura que, de soporte a la definición de particularidades del dominio, tanto en la supervisión centralizada del estado de las aplicaciones como en la toma de decisiones y acciones asociadas ante situaciones sobrevenidas, y menos aún si la intervención es a nivel de aplicación. Probablemente, pueda deberse a que la gestión de las aplicaciones se ha entendido habitualmente como la gestión individual de un conjunto de componentes interrelacionados. Sin embargo, estos componentes deben considerarse como un todo para permitir garantizar el cumplimiento de los requisitos a nivel de aplicación. En el caso particular de la adaptabilidad, también se debe considerar la aplicación como un conjunto cuando, para hacer frente a eventos inesperados, una aplicación debe actuar

sobre otra (iniciarla/detenerla o cambiar sus parámetros de ejecución...). Por ejemplo, si en un paciente que está monitorizado de forma remota se detecta que sus constantes vitales no están dentro de los rangos esperados, se debería poder poner en marcha nuevas tareas de supervisión para determinar la criticidad de la situación. Por lo tanto, para poder supervisar la ejecución de una aplicación, la plataforma debe saber qué es una aplicación (entidades que la forman y estado en el que se encuentran) y cómo debe cambiar su estructura en caso de que se detecten cambios en el entorno. Existen varias propuestas en la literatura que han intentado dar solución a las demandas dependientes del dominio, pero, en general son soluciones ad-hoc que difícilmente pueden aplicarse a otros ámbitos. Algunas se basan en el Desarrollo Basado en Modelos (*Model Driven Development, MDD*) (Mohamed et al., 2021), ya que el uso de modelos proporciona la abstracción necesaria para lograr una propuesta genérica. Pero no consideran un concepto de aplicación y/o definición de estado del sistema.

1.2 Estado del arte

El objetivo principal de una plataforma de gestión de aplicaciones es garantizar que las aplicaciones se ejecutan según lo especificado. Como se comentó anteriormente, las aplicaciones sensibles al contexto presentan requisitos comunes y requisitos específicos del dominio. De ellos, se pueden generalizar los requisitos principales que debe cumplir una plataforma. La Tabla 1 recoge los requisitos de la plataforma, que se pueden dividir en tres grupos: operacionales (R1-R3), que abordan la ejecución de las aplicaciones; y no operacionales, relacionados con la seguridad (R4) y la flexibilidad (R5-R8).

Desde el punto de vista operacional, las aplicaciones, desplegadas en dispositivos heterogéneos, realizan tareas de adquisición de datos, procesamiento y actuación. Por lo tanto, la plataforma debe permitir la ejecución distribuida de estas tareas, así como la comunicación entre ellas (*R1: ejecución y comunicación distribuida*). Además, es

necesario el soporte para un despliegue eficiente, teniendo en cuenta los recursos disponibles y su demanda por parte de las aplicaciones (*R2: despliegue eficiente de aplicaciones*). Sumado a esto, los sistemas sensibles al contexto están compuestos por un conjunto de aplicaciones dinámico en número y tamaño, con requisitos temporales, cuyo inicio, parada y funcionamiento normal deben ser controlados por la plataforma de gestión de aplicaciones (*R3: gestión del ciclo de vida*).

Tabla 1: Requisitos de la plataforma

Tipo	Requisito	Descripción
Operacional	R1	Ejecución distribuida y comunicación.
	R2	Despliegue eficiente de la aplicación.
	R3	Gestión del ciclo de vida.
Seguridad (no operacional)	R4	Seguridad.
Flexibilidad (no operacional)	R5	Auto-adaptabilidad.
	R6	Trazabilidad / Auto-conciencia (<i>self-awareness</i>).
	R7	Auto-recuperación (<i>self-healing</i>).
	R8	Variabilidad de dominio.

En cuanto a los requisitos no operacionales, la seguridad del sistema requiere de mecanismos para asegurar la privacidad, confidencialidad, autenticidad e integridad de los datos (*R4: seguridad*). Es importante remarcar que las aplicaciones sensibles al contexto se incluyen dentro de los llamados sistemas auto-adaptativos, por lo que también requieren de capacidad para adaptarse de forma autónoma a los cambios en su entorno. Esto implica no solo conocer el contexto (*context-awareness*), sino también la auto-conciencia (*self-awareness*) (Krupitzer et al., 2015). Para lograr la adaptación al contexto, la plataforma debe estar dotada de mecanismos de reconfiguración dinámica activados por aplicaciones y que le permitan reaccionar ante situaciones relevantes, cambiando su comportamiento (*R5: auto-adaptabilidad*). La auto-conciencia implica ser consciente de la disponibilidad de recursos de manera dinámica. Para ello, la plataforma debe realizar un seguimiento tanto del estado de los recursos de

infraestructura como del estado de las aplicaciones (*R6: trazabilidad / auto-conciencia (self-awareness)*).

En cuanto a la disponibilidad de las aplicaciones, la plataforma debe minimizar las interrupciones del servicio, incluyendo capacidad para la detección de fallos y la recuperación automática del servicio, manteniendo el estado de las aplicaciones (*R7: auto-recuperación (self-healing)*).

Finalmente, cada dominio de aplicación tiene sus particularidades en términos de especificación de aplicaciones (qué conceptos definen las aplicaciones y sus relaciones) y gestión de ejecución, o incluso en lo referido a tipos de recursos. Para aprovechar el gran esfuerzo que supone el diseño y desarrollo de una plataforma de gestión, sería beneficioso disponer de una plataforma personalizable a diferentes dominios (*R8: variabilidad de dominio*).

Las siguientes secciones analizan el trabajo relacionado que aborda los requisitos identificados. La Tabla 2 recoge el análisis de las principales plataformas de gestión.

1.2.1 Requisitos Operacionales (R1-R3)

Las arquitecturas de software distribuidas consideran las aplicaciones como un conjunto de módulos (unidades computacionales) que se ejecutan en diferentes nodos e interactúan para lograr la funcionalidad de la aplicación. Sin embargo, la definición del módulo y su composición difieren de una arquitectura a otra. Por ejemplo, en la Ingeniería del Software Basada en Componentes (*Component-Based Software Engineering, CBSE*) (Vale et al., 2016), los componentes se desarrollan como cajas negras que ofrecen servicios de forma independiente a la aplicación. Las aplicaciones son composiciones de componentes en base a su interfaz o siguiendo un modelo de componentes. Las aplicaciones basadas en multi-agentes (MAS) consisten en componentes software inteligentes y débilmente acoplados, llamados agentes, que son autónomos (toman decisiones sin intervención humana directa), reactivos (reaccionan

a cambios en su contexto) y sociales (interactúan entre sí, cooperando o compitiendo)(Michael Wooldridge, 2009). En la Computación Orientada a Servicios (*Service Oriented Computing*, SOC), las unidades computacionales se denominan servicios. Los proveedores publican los servicios en repositorios como cajas negras que los consumidores pueden descubrir y usar, o incluso componer, creando nuevos servicios (Al-Jaroodi and Mohamed, 2012). En los últimos años, la aparición del estilo arquitectónico de microservicios ha permitido la construcción de aplicaciones distribuidas altamente escalables, en base a servicios pequeños y poco acoplados que se comunican a través de protocolos ligeros (James Lewis and Martin Fowler, 2014). Las tecnologías de organización en contenedores que posibilitan una virtualización ligera se han convertido en el estándar de facto para empaquetar microservicios en la nube (Wang et al., 2021).

Existen aproximaciones que, basándose en un estilo arquitectónico distribuido, están orientadas a facilitar el desarrollo y/o la gestión de aplicaciones. Suelen proporcionar mecanismos para desplegar, comunicar y gestionar el ciclo de vida de los módulos de la aplicación. Es el caso, por ejemplo, de *Java Agent Development (JADE)* (Bellifemine et al., 2008), la implementación más utilizada del estándar *Foundation for Intelligent Physical Agents (FIPA)* (Foundation for Intelligent Physical Agents, 2005) para MAS; y de Kubernetes (Kubernetes, 2022), la herramienta más extendida para la orquestación de aplicaciones basadas en microservicios implementadas mediante contenedores.

Cualquier plataforma construida mediante cualquiera de estas aproximaciones o construida directamente sobre una arquitectura de software distribuida cumple directamente con el requisito R1 (ejecución distribuida y comunicación) y con , al menos, una versión básica de R2 (despliegue eficiente de aplicaciones) y R3 (gestión del ciclo de vida).

Tabla 2: Cumplimiento de requisitos por las plataformas de gestión de aplicaciones.

PLATAFORMA	R1-R3: REQ. OPERACIONALES	R5: AUTO-ADAPTABILIDAD		R6: TRAZABILIDAD / AUTO-CONCIENCIA		R7: AUTO-RECUPERACIÓN			R8: VARIABILIDAD DE DOMINIO	
		Conducido por aplic.	Aplic. objetivo de acción	Foco	Modelo dinámico	Transparente para aplic.	Integridad estado	Fallo nodo	Concepto aplic.	Genérica
(Khan et al., 2008)	CBSE	No	No	CC	No	---	---	---	No	Sí
ACCADA (Gui et al., 2011)	CBSE	No	No	CC	No	---	---	---	No	Sí
MUSIC (Hallsteinsen et al., 2012)	CBSE + SOC	No	No	CC	No	No	Sí	No	No	Sí
DARE (Albassam et al., 2017)	CBSE	No	No	CC	Sí	Sí	Sí	No	No	Sí
(Hussein et al., 2011)	CBSE	No	No	CC	Sí	---	---	---	No	Sí
THOMAS / PANGEA (Argente et al., 2011) / (Villarrubia et al., 2017)	MAS + SOC	Restringido	Sí	CO	Sí	Sí	No	No	No	Sí
iLAND (Garcia Valls et al., 2013)	SOC	No	No	CC	Sí	Sí	No	Sí	Sí	No
DAMP (Agirre et al., 2016)	CBSE	Sí	Sí	CA	Sí	Sí	Sí	Sí	Sí	No
EI4MS (He et al., 2021)	Microservicios	No	No	CQU	Sí	Sí	Sí	No	No	Sí

CC = Centrado-Componente, CO = Centrado-Organización, CA = Centrado-Aplicación, CQU = Centrado-QoS-Usuario

1.2.2 Requisitos No-Operacionales

Las plataformas distribuidas se pueden extender con mecanismos que permitan cumplir requisitos no operacionales. Las siguientes subsecciones analizan lo investigado en este sentido.

1.2.2.1 Seguridad (R4)

En (Hassija et al., 2019) se puede encontrar una completa investigación acerca de mecanismos que permitan asegurar el acceso, almacenamiento, procesamiento y transmisión de datos. Las soluciones más ampliamente aplicadas son, entre otras, infraestructuras de clave pública (*Public Key Infrastructure*, PKI), mecanismos de autenticación y autorización, encriptación, sockets seguros (*Secure Socket Layer*, SSL) y la tecnología “blockchain”. Sin embargo, todas estas soluciones pueden incluirse en las plataformas de forma transparente a la gestión de las aplicaciones, por lo que no se consideran dentro del alcance de la presente tesis doctoral.

1.2.2.2 Auto-adaptabilidad (R5)

La auto-adaptabilidad generalmente se basa en la implementación de modelos de bucles MAPE-K (es decir, hacer uso en la computación autónoma del concepto de bucle de retroalimentación de la teoría de control) (Arcaini et al., 2015). Según (Khabou et al., 2014), la auto-adaptabilidad es una tarea compleja que se puede dividir en cuatro fases: (1) monitorización/recolección de parámetros del contexto; (2) detección de cambios relevantes mediante el análisis de los datos recopilados; (3) planificación de acciones de adaptación apropiadas para responder a los cambios; (4) ejecución de las acciones previstas.

Las dos primeras fases dependen en gran medida del contexto específico, que varía de un campo a otro (Li et al., 2015). Algunos trabajos de investigación se han centrado en la especificación del contexto. Por ejemplo, en (Iqbal et al., 2021) se presenta un modelo de contexto basado en ontologías, que permite mejorar las capacidades de interacción de los distintos usuarios de dispositivos móviles adaptativos. Modela el contexto a través de sus cuatro elementos principales: el dispositivo, el usuario, el entorno (ubicación y tiempo) y la actividad realizada por el usuario. Estos cuatro aspectos también son tenidos en cuenta en el modelo de contexto propuesto en (Boudaa et al., 2018), dentro del ámbito de las ciudades inteligentes, que permite a un sistema de recomendación proponer servicios relevantes para los usuarios. Otros trabajos, en cambio, consideran la monitorización del contexto como parte de la funcionalidad de la aplicación, ya que son los propios módulos de la aplicación los encargados de capturar y procesar información del contexto, siendo por ello innecesaria su caracterización. Esta última es la concepción adoptada en esta tesis doctoral.

En cuanto a la fase 3 (planificación de acciones de adaptación apropiadas para responder a los cambios), se han propuesto varias técnicas para seleccionar acciones de adaptación. Una de ellas es expresar la auto-adaptabilidad a través de la variabilidad de la aplicación (Galster et al., 2014), definida en base a puntos de variación (dónde puede ocurrir un cambio planificado) y variantes (qué opciones se pueden seleccionar). En (Khan et al., 2008) cada componente se considera como un punto de variación y las implementaciones como las variantes correspondientes, cada una relacionada con una situación específica en el contexto. Otros trabajos hacen uso de reglas para la toma de decisiones, una de las soluciones más utilizadas, ya que proporcionan un método de clasificación fácil y automatizable (Ramírez et al., 2021). Por ejemplo, (Hussein et al., 2011) y (Rocha et al., 2013) utilizan reglas para detectar y clasificar situaciones relevantes, siendo posible razonar la mejor respuesta.

Las acciones a ejecutar en la fase 4 comprenden desde propuestas fijas y ad-hoc, tales como simples avisos (Hsu and Nieh, 2020; Rocha et al., 2013) o alarmas (Baek et al., 2021), hasta otras más flexibles, basadas en mecanismos de reconfiguración dinámica (Agirre et al., 2016; Argente et al., 2011). En el caso de la reconfiguración dinámica, una entidad externa automatiza y gestiona la ejecución de la adaptación, separando la lógica de adaptación de la lógica de aplicación. La reconfiguración dinámica se ha aplicado en dos niveles: componente y aplicación. La mayoría de los enfoques funcionan a nivel de componente, siendo posible agregar, quitar, reemplazar y/o volver a conectar módulos de aplicación. En ocasiones, la adaptación está restringida a solicitudes externas, como en la plataforma DARE (Albassam et al., 2017), lo que limita la autonomía de las aplicaciones. Otras veces, la propia plataforma es la responsable de detectar cambios de contexto y seleccionar las implementaciones de componentes que mejor se ajusten a un nuevo estado del contexto, como en el proyecto MUSIC (Hallsteinsen et al., 2012) y la plataforma presentada en (Khan et al., 2008). De forma similar, la arquitectura EI4MS (He et al., 2021) es capaz de detectar la degradación en la calidad de servicio percibida por el usuario y calcular un plan de evolución óptimo, que posteriormente es ejecutado por los propios microservicios que la arquitectura gestiona. Pero, en todos estos casos la plataforma debe ser consciente de su contexto, para ser capaz de detectar cambios en él. En el caso de la plataforma ACCADA (Gui et al., 2011) son posibles las dos opciones: permite que usuarios externos soliciten el reemplazo de un componente para su actualización, pero también dispone de un módulo de sistema capaz de detectar cuándo se violan determinadas situaciones de contexto, iniciando, en ambos casos, el correspondiente proceso de reconfiguración dinámica.

Como la adaptación a nivel de componente no cubre el concepto de aplicación, se han propuesto aproximaciones para extender la reconfiguración dinámica al nivel de aplicación. La plataforma THOMAS combina tecnologías orientadas a agentes y servicios para permitir organizaciones estructuradas de agentes (Argente et al., 2011). En este caso, la reconfiguración dinámica implica la incorporación de nuevas

estructuras organizacionales, así como la inclusión o eliminación de miembros. Sin embargo, estas capacidades están restringidas a ciertos roles de agente. En el middleware iLAND (Garcia Valls et al., 2013), la reconfiguración dinámica consiste en una recomposición acotada en el tiempo de las aplicaciones basadas en servicios durante su ejecución, siendo iniciada por el propio middleware cuando las aplicaciones se arrancan o detienen. Otros trabajos van un paso más allá, permitiendo que los componentes soliciten acciones de adaptación dirigidas a toda la aplicación. Este es el caso de la plataforma DAMP (Agirre et al., 2016) donde es posible arrancar y parar aplicaciones, pero sólo si han sido previamente desplegadas, con el consecuente consumo de recursos.

1.2.2.3 Trazabilidad/Auto-conciencia (*self-awareness*) (R6)

Algunas de las plataformas de gestión analizadas rastrean el estado del sistema para tomar las decisiones más adecuadas en tiempo de ejecución. La mayoría utiliza algún tipo de repositorio, que varía de una plataforma a otra. Por ejemplo, el framework DARE (Albassam et al., 2017) dispone de un mapa de configuración con el mapeo de los componentes en ejecución a los nodos, que se calcula automáticamente mediante técnicas de sondeo o *gossip*. Las plataformas (Khan et al., 2008), (Gui et al., 2011) y (Hallsteinsen et al., 2012) toman decisiones en tiempo de ejecución basadas en modelos de adaptación proporcionados durante el diseño. Estos modelos contienen alternativas de implementación según diferentes valores de contexto. Los algoritmos de composición del middleware iLAND (Garcia Valls et al., 2013) soportan un modelo de aplicación configurado con parámetros de calidad de servicio (*Quality of Service*, QoS) sobre el procesamiento de datos y la necesidad de recursos por parte de los servicios de la aplicación. El modelo de sistema usado en EI4MS (He et al., 2021) para la construcción de los planes de evolución describe el estado de despliegue actual del sistema mediante la siguiente información: servicios lógicos existentes, nodos cloud/edge disponibles, demandas del usuario, etc. En (Hussein et al., 2011), el modelo de contexto está separado del modelo de sistema, pero ambos incluyen aspectos

dinámicos que los relacionan en tiempo de ejecución. Concretamente, cada componente dispone de un conjunto de funciones con las cuales es capaz de calcular la implementación de servicio que mejor se adapta a cada situación de contexto. Cabe mencionar que todos estos trabajos consideran una aplicación como un grafo de componentes que interactúan, cada uno de los cuales puede tener varias realizaciones o implementaciones, excepto el último trabajo, (Hussein et al., 2011), que permite una definición jerárquica de los componentes. Es más, todas estas propuestas están centradas en los componentes y no consideran el concepto de aplicación, entendido como un conjunto de componentes interrelacionados que deben ser gestionados como un todo. Como resultado, no proporcionan una gestión de la ejecución centrada en las aplicaciones que pueda cubrir demandas a nivel de aplicación.

Existen aproximaciones que intentan definir estructuras de aplicación más complejas. En el contexto de MAS, se han propuesto metodologías de ingeniería orientada a agentes que tienen en cuenta aspectos sociales para el llamado *open-MAS* (Coutinho et al., 2019): un conjunto dinámico de agentes, que pueden ser proporcionados por diferentes desarrolladores, cada uno de ellos con un interés propio. En concreto, estas metodologías permiten especificar sociedades de agentes u organizaciones de agentes, que desempeñan diferentes roles, y cuyas interacciones están conducidas mediante un conjunto de reglas, normas y restricciones (Gómez-Sanz and Fuentes-Fernández, 2015; Isern et al., 2011). Partiendo de esta idea de sociedades de agentes, los autores en (Argente et al., 2011) y (Zato et al., 2013) proponen plataformas para la gestión en tiempo de ejecución de organizaciones virtuales dinámicas en *open-MAS*. Proporcionan facilidades para que los agentes ingresen o abandonen voluntariamente una organización virtual, así como también para la creación, eliminación y modificación de organizaciones virtuales bajo demanda. Sin embargo, es precisamente el carácter individualista de este tipo de agentes lo que imposibilita la gestión centrada en aplicaciones. La plataforma DAMP (Agirre et al., 2016) considera una aplicación como una entidad única para lograr el cumplimiento de QoS de la aplicación. Proporciona un servicio de middleware que permite el registro de aplicaciones antes de su ejecución.

Esta información, junto con la supervisión de la disponibilidad de recursos, se utiliza en tiempo de ejecución para reconfigurar las aplicaciones cuando es necesario.

1.2.2.4 Auto-recuperación (*self-healing*) (R7)

Se entiende por auto-recuperación la capacidad del sistema para detectar cuándo un servicio deja de estar disponible y restaurarlo. Puede ser un proceso proactivo y/o reactivo (Psaier and Dustdar, 2011). Los sistemas de recuperación proactivos implican prevención, es decir, detectar la degradación del servicio antes de que falle. Las tareas de prevención se pueden tener en cuenta desde la fase de diseño, como un caso particular de auto-adaptabilidad, e incluso es posible decidir el momento óptimo en el que actuar. Por el contrario, la auto-recuperación reactiva es un desafío mayor, ya que interviene cuando ya se ha producido el fallo y debe permitir la recuperación del servicio ante problemas repentinos y no previsibles, tanto en componentes como en nodos. A veces, incluso es necesario mantener la coherencia del estado de la aplicación.

Los mecanismos de detección de fallos generalmente se basan en los llamados mensajes de latido (*heartbeat messages*). Se distinguen dos enfoques: 1) sondeo o *gossip*: una entidad solicita a componentes o nodos confirmación de su estado (Huan and Hidenori, 2012); y 2) prueba de vida: un componente o nodo informa de su estado. Algunos trabajos proponen una gestión centralizada de mensajes de latido a través de un módulo de plataforma encargado de detectar fallos en nodos. Por ejemplo, la plataforma DAMP (Agirre et al., 2016) y el middleware iLAND (Garcia Valls et al., 2013) utilizan técnicas de prueba de vida, mientras que (Ruiz et al., 2015) se basa en mecanismos de sondeo. Otro ejemplo de propuesta centralizada se presenta en (Vayghan et al., 2021), donde se hace uso de la interfaz de programación (*Application Programming Interface, API*) de Kubernetes para monitorizar eventos de servicios que puedan indicar fallos en los pods. También existen propuestas descentralizadas que mejoran la autonomía y la capacidad de detección de fallos de los sistemas. En el framework DARE (Albassam et al., 2017), cada nodo alberga un módulo encargado de

sondear y reportar posibles fallos de nodo, mientras que en (Huan and Hidenori, 2012) todos los nodos se encargan de revisar el correcto funcionamiento del resto.

En cuanto a la recuperación de los fallos, los trabajos de (Hallsteinsen et al., 2012) y (García-Magariño and Gutiérrez, 2013) dan soporte a la recuperación reactiva a través de la propia programación de los componentes. Sin embargo, aunque la inclusión de código específico en los módulos de la aplicación permite una rápida detección y recuperación ante fallos, la lógica de la aplicación debe garantizar su propia disponibilidad. Para evitar esta dependencia, las plataformas habitualmente han proporcionado gestión de réplicas de manera transparente a la aplicación (Guerraoui and Schiper, 1997).

Otros trabajos proponen una entidad central con una visión global del sistema. Este enfoque permite la separación de la lógica de recuperación de la lógica de la aplicación y permite tomar decisiones más adecuadas. Por ejemplo, (Agirre et al., 2016) y (Garcia Valls et al., 2013) utilizan algoritmos de recomposición para seleccionar la mejor réplica, mientras que las arquitecturas de (Ruiz et al., 2015) y (Albassam et al., 2017) cuentan con un módulo específico para determinar si un fallo se puede recuperar o no. La consistencia del estado de la aplicación es un aspecto relevante a la hora de recuperar un fallo inesperada, ya que asegura la continuidad del servicio. Se pueden encontrar dos enfoques principales (Funk et al., 2007). Por un lado, la recuperación basada en puntos de control permite la reversión del sistema a su estado coherente más reciente (Huan and Hidenori, 2012). Para ello, no solo es necesario almacenar el estado del sistema, sino también los mensajes recibidos entre los diferentes puntos de control. Por otro lado, una solución más fácil y flexible es proporcionar mecanismos para transferir y restaurar el estado de la aplicación. En (Agirre et al., 2016) los componentes envían periódicamente su estado a la plataforma, que lo almacena para su posible restauración en las implementaciones seleccionadas. Sin embargo, esto provoca una sobrecarga en la plataforma y solo es posible para componentes periódicos. En la propuesta realizada en (Vayghan et al., 2021), el componente llamado

State Controller está integrado en Kubernetes para permitir la recuperación de pods con estado. Aunque esta propuesta tiene en cuenta la elasticidad (es decir, que varios pods ofrezcan el mismo servicio), la transferencia del estado está limitada entre los integrantes de parejas de pods.

1.2.2.5 Conclusiones y variabilidad de dominio (R8)

En conclusión, para cumplir con los requisitos no operacionales, es necesaria una plataforma de gestión que extienda la implementación de una arquitectura de software distribuida. La reconfiguración dinámica es el mejor mecanismo para lograr la auto-adaptabilidad y la recuperación ante fallos. En ambos casos, los enfoques descentralizados mejoran la autonomía del sistema, disminuyendo la sobrecarga de la plataforma. Sin embargo, se necesita una visión global de todo el sistema para tomar las decisiones que mejor se adapten a las necesidades de todas las aplicaciones en ejecución, en un momento concreto. Al analizar los trabajos de la literatura se observa que las plataformas de gestión suelen centrarse en alguno de los requisitos identificados, pero no los cubren todos. Además, no existe una gestión de la ejecución centrada en las aplicaciones, ya que la mayoría de las propuestas ni siquiera tienen en cuenta a las aplicaciones como entidad. Y, si lo hacen, las consideran como una estructura ad-hoc y/o fija. Lograr una gestión centrada en la aplicación requiere que la plataforma sea consciente del concepto de aplicación en un dominio específico. Una definición ad-hoc de las aplicaciones hace que la correspondiente plataforma de gestión sea también una solución ad-hoc (Agirre et al., 2016; Garruzzo et al., 2007; Rosaci and Sarné, 2006). Adaptar estas plataformas a otros dominios implica rediseñarlas y/o volver a implementarlas, como el caso de la arquitectura MASHA que inicialmente fue desarrollada para sitios web (Rosaci and Sarné, 2006) y posteriormente fue adaptada a sistemas de eLearning (Garruzzo et al., 2007). Por lo tanto, tener una arquitectura genérica y personalizable reduciría o incluso evitaría la necesidad de este arduo trabajo (R8: *Variabilidad de dominio*). Para ello, es fundamental poder definir la estructura de la aplicación de forma abstracta.

1.3 Objetivos

Esta sección describe el objetivo principal y los objetivos parciales de la presente tesis, relacionándolos con las publicaciones en las que se han acometido (ver Tabla 3).

0.0. El objetivo general del presente trabajo se centra en la definición de una plataforma genérica, personalizable y extensible para la gestión de aplicaciones sensibles al contexto, que, además de los servicios básicos de gestión, también cubra las demandas de flexibilidad de este tipo de aplicaciones.

Con el fin de lograr este objetivo principal, se plantean los siguientes objetivos parciales que abarcan diferentes aspectos relacionados con las aplicaciones sensibles al contexto, la gestión de su ejecución y la personalización de la plataforma a dominios concretos:

0.1. Identificar los requisitos de gestión de ejecución de las aplicaciones sensibles al contexto: análisis de aplicaciones pertenecientes a diferentes dominios para identificar sus requisitos tanto funcionales como no funcionales, así como comunes y específicos de dominio.

0.2. Gestión de la ejecución centrada en las aplicaciones:

0.2.1. Identificar e incluir el concepto de aplicación en la plataforma resulta clave para poder considerar la aplicación como una entidad a gestionar y cubrir aquellos requisitos que afectan al conjunto de una aplicación.

0.2.2. Dotar a la plataforma de mecanismos que favorezcan la inteligencia distribuida y que permitan la descentralización de la toma de decisiones, mejorando la autonomía del sistema.

- 0.2.3.** Dotar a la plataforma de mecanismos para la gestión del estado del sistema, que permitan trazar la ejecución de las aplicaciones y gestionar eficientemente los recursos del sistema, consiguiendo la supervisión centralizada de la inteligencia distribuida.
 - 0.2.4.** Dotar a la plataforma de mecanismos de reconfiguración dinámica para hacer frente a las necesidades de flexibilidad de las aplicaciones. Esta reconfiguración puede ser conducida por la propia aplicación (auto-adaptabilidad) o conducida por la propia plataforma de forma transparente a la aplicación (disponibilidad, QoS flexible...).
- 0.3.** Personalización y extensión de la plataforma a dominios concretos:
- 0.3.1.** Definir de forma genérica el concepto de aplicación, así como el estado del sistema y la interfaz para su gestión.
 - 0.3.2.** Identificar el núcleo de la plataforma común a todos los dominios.
 - 0.3.3.** Establecer una metodología de personalización para hacer frente a las particularidades del dominio.
 - 0.3.4.** Definir mecanismos de extensión de la plataforma que permitan hacer frente a QoS específica de dominio y/o aplicación.

1.4 Resultados y Discusión

El trabajo desarrollado a lo largo de esta tesis doctoral ha dado lugar a un conjunto de resultados que han sido publicados en revistas con índice de impacto y presentados tanto en conferencias internacionales de reconocido prestigio en el campo de la investigación como en conferencias nacionales. De entre todos ellos, se han seleccionado los 5 más importantes que permitan describir en detalle los resultados alcanzados, relacionándolos con los objetivos planteados (ver Tabla 3):

- P.1.** Armentia, A., Gangoiti, U., Priego, R., Estévez, E., y Marcos, M. (2015). Flexibility Support for Homecare Applications Based on Models and Multi-Agent Technology. *Sensors*, 15 (12), pp. 31939–31964. <https://doi.org/10.3390/s151229899>.
- P.2.** Priego, R., Iriondo, N., Gangoiti, U., y Marcos, M. (2017). Agent-based middleware architecture for reconfigurable manufacturing systems. *The International Journal of Advanced Manufacturing Technology*, 92 (5), pp. 1579–1590. <https://doi.org/10.1007/s00170-017-0154-z>.
- P.3.** Armentia, A., Gangoiti, U., Orive, D., y Marcos, M. (2017). Dynamic QoS Management for Flexible Multimedia Applications. In: 20th IFAC World Congress, Toulouse, France. *IFAC-PapersOnLine*, 50 (1), pp. 5920–5925. <https://doi.org/10.1016/j.ifacol.2017.08.1483>.
- P.4.** Gangoiti, U., López, A., Armentia, A., Estévez, E., y Marcos, M. (2021). Model-Driven Design and Development of Flexible Automated Production Control Configurations for Industry 4.0. *Applied Sciences*, 11 (5), 2319, pp. 1–27. <https://doi.org/10.3390/app11052319>.
- P.5.** Gangoiti, U., López, A., Armentia, A., Estévez, E., Casquero, O., y Marcos, M. (2022). A customizable architecture for application-centric management of context-aware applications. *IEEE Access*, 10, pp. 1603-1625. <https://doi.org/10.1109/ACCESS.2021.3138586>.

Es importante señalar que, de estas 5 publicaciones, sólo 3 conforman el compendio de la presente tesis doctoral. En concreto: P.1, P.4 y P.5.

Tabla 3: Relación objetivos - publicaciones

Objetivo	Publicación
O.0	P.5
O.1	P.1, P.2, P.3, P.4

Objetivo		Publicación
0.2	0.2.1	P.1, P.2, P.3, P.4
	0.2.2	P.1, P.2, P.3, P.4
	0.2.3	P.1, P.2, P.3
	0.2.4	P.1, P.2, P.4
0.3	0.3.1	P.1, P.5
	0.3.2	P5
	0.3.3	P.1, P.4, P.5
	0.3.4	P3, P.5

A lo largo de los siguientes apartados se describen los resultados de la presenta tesis doctoral en referencia al cumplimiento de los requisitos de plataforma identificados (ver Tabla 1). Se detallarán los artículos donde se ha trabajado cada aspecto, así como con qué objetivo de la tesis doctoral tiene relación. Resulta importante recordar que el requisito de seguridad (R4) se considera fuera del ámbito de este trabajo.

1.4.1 Inteligencia Distribuida: Requisitos Operacionales (R1-R3)

La presente tesis doctoral ha sido un trabajo de apoyo tanto para otras tesis doctorales como para proyectos de investigación realizados en diferentes dominios. Así, a lo largo de su desarrollo se han propuesto diferentes plataformas de dominio ad-hoc que han permitido identificar los requisitos comunes a todos los campos y trabajar soluciones que permiten su cumplimiento. Del mismo modo, también ha sido posible identificar y abordar particularidades propias de cada dominio (objetivo 0.1).

Las figuras Figura 1, Figura 2 y Figura 3 muestran las diferentes arquitecturas propuestas en estos trabajos. Inicialmente, en P.1 se propuso la arquitectura mostrada en la Figura 1, que se corresponde con el dominio de la monitorización remota de la salud (*eHealthcare*, eHC). Es importante señalar que esta misma arquitectura de plataforma se pudo emplear para la gestión de aplicaciones para la detección de intrusos en el campo de los sistemas de video-vigilancia, tal y como se presenta en P.3. Sin embargo, para el caso de los sistemas de fabricación flexible (FMS) que dieron lugar a P.2 y P.4 (ver Figura 2) la plataforma inicial tuvo que ser rediseñada en su mayor parte, ya que la estructura de la aplicación, los recursos y la gestión de la aplicación eran diferentes. Así, a pesar de compartir una arquitectura base similar (resaltada en un recuadro verde en la Figura 2), fue necesario incluir nuevos elementos encargados de la gestión de la aplicación (*Plant_AAS Agent*, que hacen las veces del *Application Manager* de la Figura 1), de la gestión de los recursos (*Controller_AAS Agent*, que complementan a los *Node Agent* de la Figura 1, y que, por simplicidad, no se muestran en la Figura 2) y de la supervisión del sistema para la recuperación de fallos de recurso específico de dominio (*QoS Manager*). Gracias a la experiencia adquirida en estos dominios ha sido posible abordar la tarea indicada por el objetivo O.0, principal aportación de P.5: la definición de una plataforma genérica, personalizable y extensible para la gestión de aplicaciones sensibles al contexto, que, además de los servicios básicos de gestión, también cubra las demandas de flexibilidad de este tipo de aplicaciones. La Figura 3 muestra la arquitectura de dicha plataforma, llamada MAS-RECON.

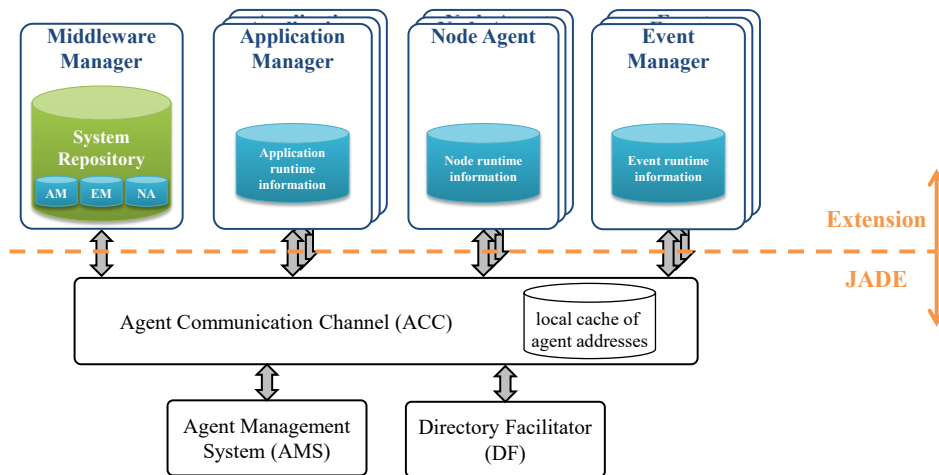


Figura 1: Arquitectura de plataforma propuesta para la gestión de sistemas de monitorización remota de la salud (eHC) en P.1 y empleada para la gestión de sistemas de video-vigilancia en P.4.

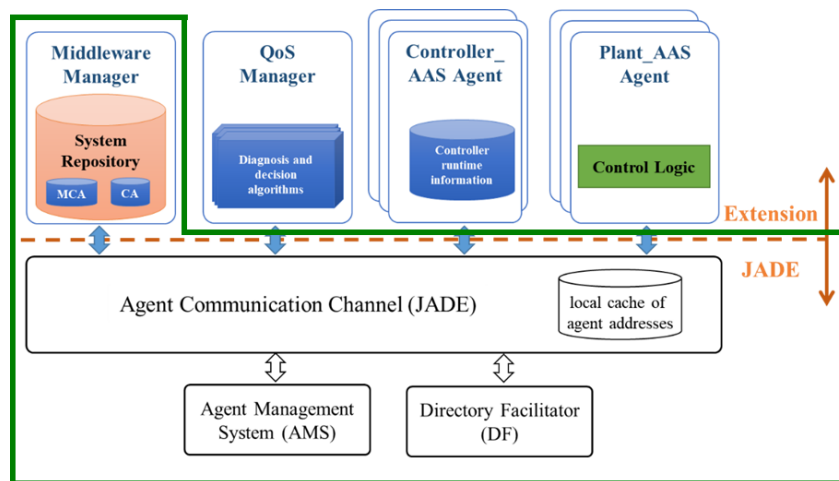


Figura 2: Arquitectura de plataforma propuesta para la gestión de sistemas de fabricación flexible (FMS), en P.2 y P.4.

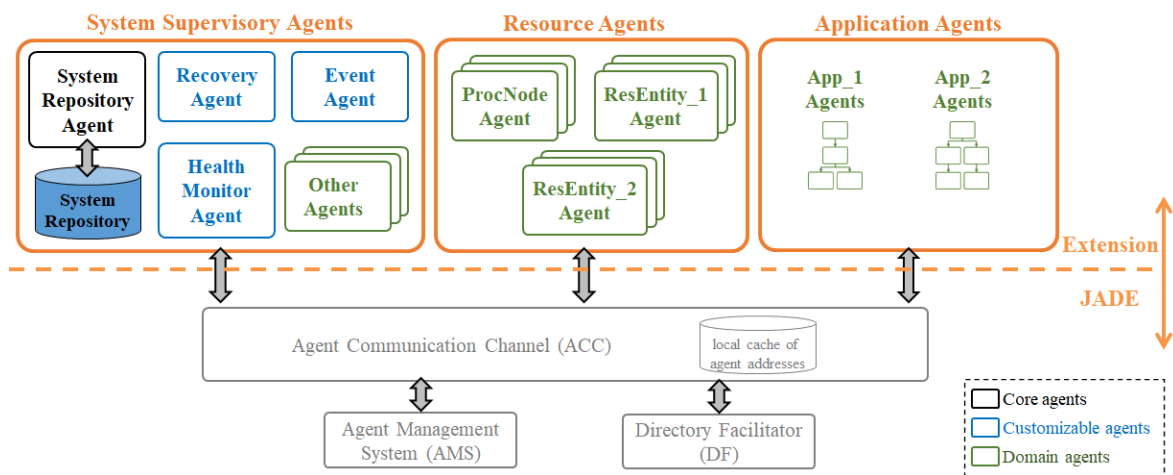


Figura 3: Arquitectura de la plataforma MAS-RECON: genérica, personalizable y extensible para la gestión de aplicaciones sensibles al contexto

Para cumplir con los requisitos operacionales (R1-R3), la plataforma se basa en tecnología multi-agente, que ha sido ampliamente utilizada para el desarrollo de sistemas complejos. Como se muestra en la Figura 3, sin pérdida de generalidad, MAS-RECON se ha construido sobre el framework JADE (Bellifemine et al., 2008), un framework de agentes compatible con FIPA, completamente desarrollado en el lenguaje de programación Java. La Fundación FIPA promueve la tecnología basada en agentes y la interoperabilidad del estándar FIPA con otras tecnologías. Cualquier infraestructura compatible con FIPA debe admitir la gestión de agentes por medio de los siguientes módulos (parte inferior de las figuras Figura 1, Figura 2 y Figura 3): el *Directory Facilitator* (DF), el *Agent Management System* (AMS) y el *Agent Communication Channel* (ACC). De acuerdo con la especificación FIPA, debe haber al menos un agente DF en la plataforma, que proporciona las páginas amarillas donde los agentes pueden registrar los servicios ofrecidos o buscar los servicios requeridos. El AMS gestiona la creación, eliminación y migración de agentes. El ACC posibilita la interoperabilidad dentro y entre diferentes plataformas. Finalmente, el llamado *Internal Platform Message Transport* (IPMT) proporciona un servicio de enrutamiento de mensajes entre agentes en una plataforma particular. Este framework JADE se ha

extendido mediante la creación de nuevos agentes, cada uno con una misión diferente (parte superior de las figuras Figura 1, Figura 2 y Figura 3).

El uso de tecnología multi-agente permite la definición de mecanismos de negociación entre agentes para la toma de decisiones distribuida, cumpliéndose el objetivo O.2.2. En las publicaciones P.1, P.2, P.3 y P.4 se han implementado diferentes propuestas de negociación acordes al dominio y objetivos perseguidos en cada una (balanceo de carga en P.1 y P.3, recuperación de la lógica de control de autómatas en fallo en P.2 y P.4). Así, en P.5 se propone la generalización de esta toma de decisiones distribuida mediante la introducción de inteligencia dentro de las entidades de dominio (*Resource Agents* y *Application Agents* en Figura 3).

1.4.2 Supervisión Centralizada: Trazabilidad (R6) y Variabilidad de Dominio (R8)

Las entidades de dominio son controladas a nivel de sistema mediante agentes de supervisión (*System Supervisory Agents* en Figura 3). Para poder llevar a cabo esta supervisión centralizada es necesario que la plataforma conozca cuál es el estado global del sistema, para lo cual debe estar almacenado en algún repositorio. Las publicaciones P.1, P.2 y P.3 abordaron el problema de la definición, almacenamiento y gestión de dicho estado en tres campos de aplicación diferentes: eHC, FMS y multimedia, respectivamente (objetivo O.2.3). Las figuras Figura 4, Figura 5 y Figura 6 muestran las diferentes estructuras de repositorio de sistema propuestas. Con el objetivo de lograr una gestión centrada en las propias aplicaciones, en estos repositorios se almacena información de las diferentes entidades que componen los sistemas, relevante para permitir su trazabilidad (objetivo O.2.1).

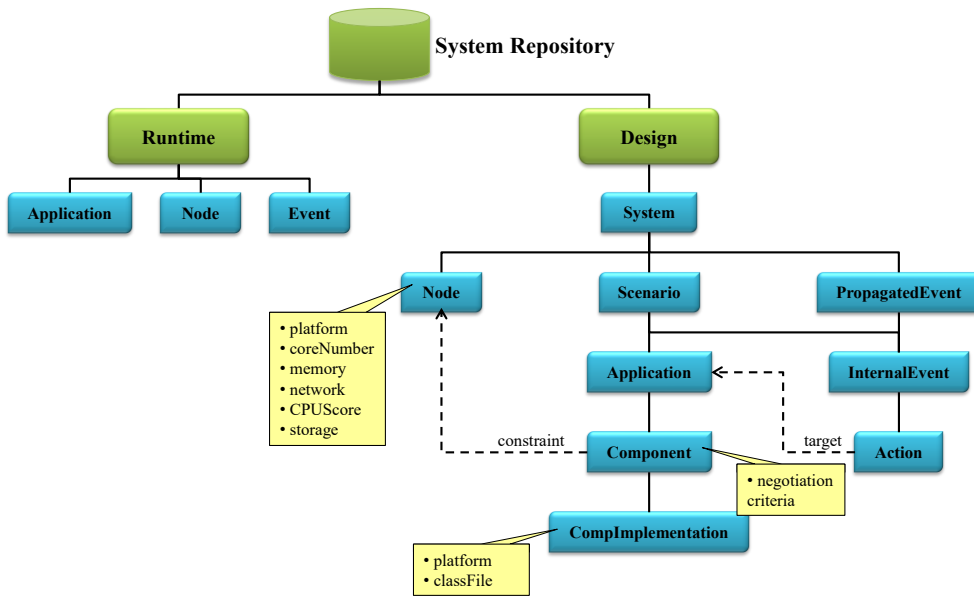


Figura 4: Estructura de repositorio para el estado global del sistema propuesta en P.1 para sistemas eHC

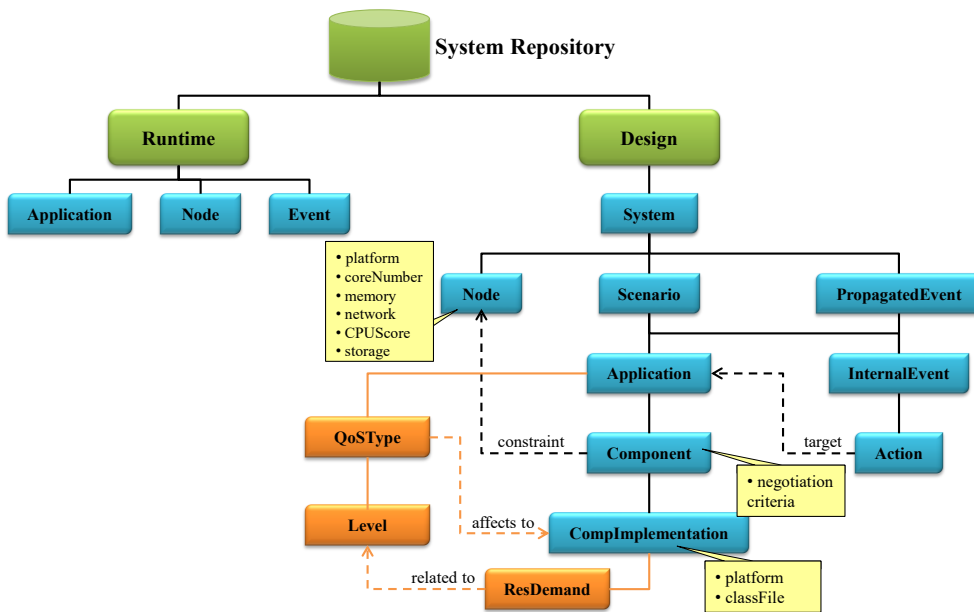


Figura 5: Estructuras de repositorio para el estado global del sistema propuesta en P.3 para sistemas multi-media.

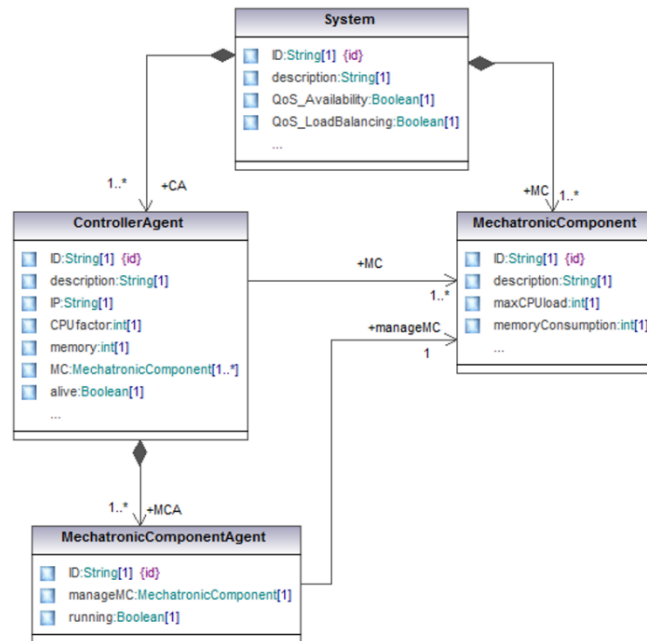


Figura 6: Estructuras de repositorio para el estado global del sistema propuesta en P.2 para sistemas FMS.

Como se puede observar, la estructura inicial propuesta para sistemas eHC (Figura 4) tuvo que ser extendida para poder gestionar la QoS dinámica propia de los sistemas multi-media (resaltado en naranja en la Figura 5). En el caso de los sistemas FMS (Figura 6), las entidades de dominio que representan al paciente (*Scenario*), a su supervisión médica (*Application*) y a los componentes de computación software que implementa dichas tareas de supervisión (*Component*) dan paso a controladores y componentes mecatrónicos (*ControllerAgent* y *MechatronicComponentAgent* en Figura 6). El objetivo de la gestión de las aplicaciones también es diferente, ya que en estos sistemas se busca la tolerancia a fallos de controlador. Por todo ello, la estructura del repositorio es totalmente diferente.

Se puede concluir que la estructura del repositorio para almacenar el estado global del sistema es totalmente dependiente del concepto de aplicación de cada dominio concreto. Por lo tanto, en P.5 se propone que la información necesaria para lograr la supervisión centralizada a nivel del sistema se almacene en el *System Repository* (SR) y que la estructura concreta de dicho repositorio esté definida por el dominio,

siguiendo el meta-modelo de la Figura 7 y haciendo referencia a las entidades que contiene: agentes recursos así como entidades y agentes de aplicación (*ResourceAgent*, *AppEntity* y *AppAgent*, respectivamente, en la Figura 7, que se corresponden con *Resource Agents* y *Application Agents* de la Figura 3). Por lo tanto, será el dominio el que defina el tipo concreto de entidades, tanto de recurso como de aplicación, sus características y las relaciones entre ellas.

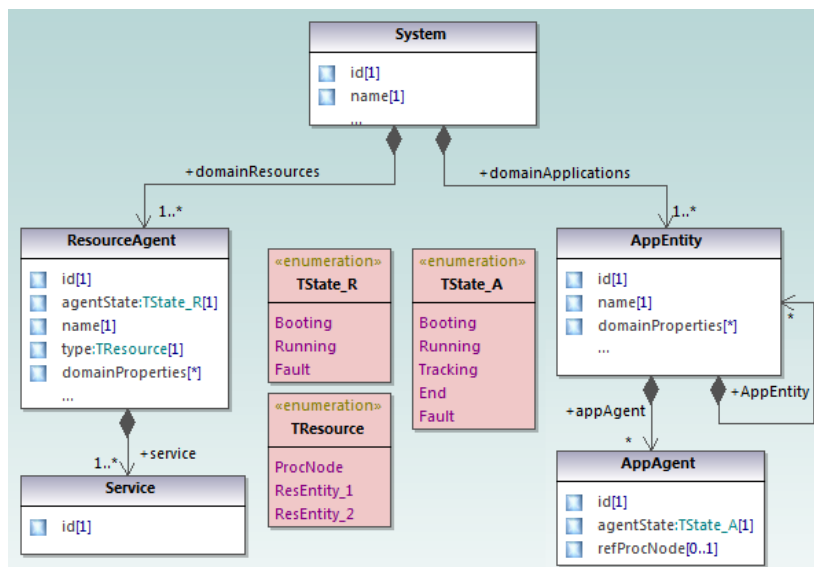


Figura 7: Meta-modelo que define la estructura genérica del repositorio de sistema (SR).

El hecho de que la estructura del SR se base en un meta-modelo de dominio permite una manipulación genérica de su contenido, tal y como se propone en P.5. Además, teniendo en cuenta que la información almacenada en el SR es aquella necesaria para la gestión de aplicaciones, tanto la común a todos los dominios como la dependiente del dominio, todos los agentes del sistema accederán al SR para operaciones de lectura y/o escritura. Por lo tanto, en P.5. también se propone que el *System Repository Agent* (SRA) sea un punto de acceso único al SR (ver Figura 3). Así, estas dos entidades, SR y SRA, constituyen el núcleo de la plataforma común a todos los dominios (objetivo O.3.2) y constituyen la base de la gestión centrada en la aplicación, contribuyendo además a cumplir con los requisitos R6 (Trazabilidad/Auto-conciencia) y R8

(Variabilidad de dominio). Además, gracias a ellos se puede concluir que MAS-RECON también cumple con el objetivo O.3.1.

En concreto, se propone que el SR sea administrado únicamente por el SRA a través de la API genérica que se muestra en la Figura 8. Esta API consta de diferentes interfaces mediante las cuales se permite: 1) el registro de recursos y aplicaciones del dominio (interfaces *iRegAgent* e *iRegApplication*, respectivamente); 2) el inicio y parada de aplicaciones (interfaz *iExecManagement*); y 3) la gestión, consulta/escritura, de propiedades de las entidades que forman las aplicaciones (interfaz *iSystemInfo*).

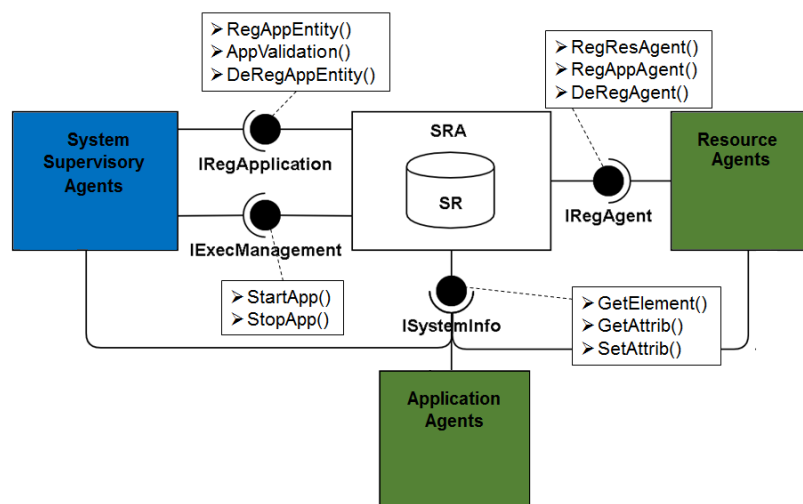


Figura 8: API genérica del System Repository Agent.

La combinación de inteligencia distribuida y supervisión centralizada permite que MAS-RECON sea capaz de extender los requisitos operacionales básicos proporcionados por JADE, permitiendo que el arranque, parada y la operación normal de las aplicaciones se adapten a las necesidades del campo de aplicación. A modo de ejemplo, la Figura 9 muestra el proceso de arranque de una aplicación. En primer lugar, las entidades de recurso registran sus agentes de recurso cuando son arrancadas (*Resources Startup* en la Figura 9), proporcionando información sobre los servicios ofrecidos por el recurso. Una vez iniciados, estos agentes realizan dos tareas. Por un

lado, supervisan el recurso físico asociado (e.g., los nodos de procesamiento pueden monitorizar la memoria disponible). Y, por otro lado, participan en procesos de negociación para decidir, de forma distribuida, el recurso más adecuado para realizar una tarea, según criterios específicos.

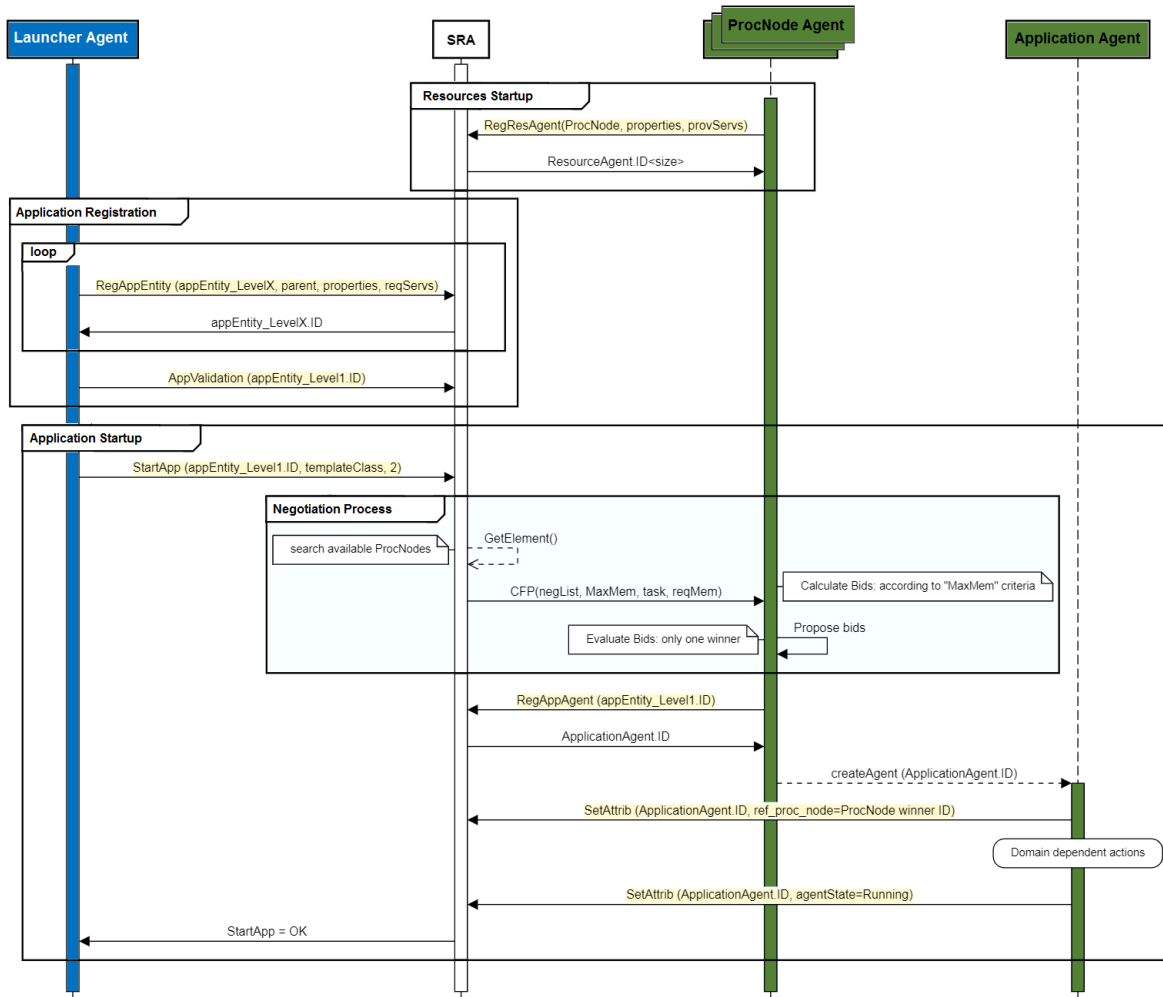


Figura 9: Diagrama de secuencia del arranque de entidades de recurso y de aplicación.

Por su parte, las aplicaciones deben registrarse antes de ser iniciadas (*Application Registration* en la Figura 9). En P.5 se propone un proceso de registro genérico que permite guardar información acerca del diseño de las aplicaciones dentro del SR, que está basado en la definición de un meta-modelo de dominio que sigue la estructura de la Figura 7. Un Agente de Supervisión de Sistema específico de dominio será el

encargado de permitir que usuarios externos puedan registrar aplicaciones. Este agente pertenece al grupo *Other Agents* de la Figura 3 y está representado por el agente *Launcher Agent* en la Figura 9. Tal y como se observa en esta figura, el proceso de registro está dividido en dos fases:

- 1) La fase inicial que consiste en el registro ordenado de todas las entidades que componen la aplicación, una a una siguiendo un orden descendente en la jerarquía de la aplicación, tal y como define el meta-modelo de dominio.
- 2) La segunda fase que comprende la validación de la aplicación contra el meta-modelo de dominio, una vez se ha completado el registro de sus entidades. Así, se garantiza que las aplicaciones registradas se ajustan a la estructura de aplicación definida para dicho dominio.

Una vez validada, se puede solicitar el arranque de la aplicación que implica el registro, la instanciación, y el despliegue de uno o más agentes de aplicación por cada entidad registrada (*Application Startup* en la Figura 9). El arranque de aplicaciones se trabajó para el dominio de eHC en P.1 y para el dominio FMS en P.2, pudiendo identificarse, en ambos casos, dos partes diferenciadas que se formalizaron en P.5:

- 1) El proceso se inicia con el arranque genérico de las entidades de aplicación de primer nivel (aquéllas que en el SR cuelgan directamente del elemento *System*), controlado por MAS-RECON. Para ello, el SRA busca los nodos de procesamiento que ofrecen los servicios requeridos por dichas entidades y lanza un proceso de negociación entre sus correspondientes Agentes de Recurso (*Negotiation Process* en la Figura 9). El proceso de negociación requiere: datos del agente, criterios de negociación y acciones a ejecutar en el nodo ganador (en este caso, registrar e instanciar el nuevo agente de aplicación).
- 2) Los agentes correspondientes a las entidades de primer nivel se encargan de iniciar la puesta en marcha de entidades de nivel inferior (*Domain dependent actions* en la Figura 9), de forma descentralizada y siguiendo un orden descendente en la jerarquía de la aplicación. Así, en esta parte del arranque, las

entidades de cada nivel son responsables del arranque de todas aquéllas de su nivel inferior. Al depender de la estructura concreta de la aplicación, se trata de una fase específica del dominio que se debe determinar en la personalización a dominio de MAS-RECON.

1.4.3 Flexibilidad: Auto-adaptabilidad (R5) y Auto-recuperación (R7)

La plataforma MAS-RECON cumple con el objetivo O.2.4 proporcionando mecanismos para hacer frente a las necesidades de flexibilidad de las aplicaciones, permitiendo dos tipos de reconfiguración dinámica:

1. Decidida por la propia aplicación, que es capaz de adaptarse a cambios en su entorno (soporte a la auto-adaptabilidad, R5)
2. Conducida por la propia plataforma, de forma transparente a la aplicación, con el objetivo de asegurar la disponibilidad de la aplicación, incluso a pesar de fallos de recurso (soporte a la auto-recuperación, R7).

En lo que se refiere a la auto-adaptabilidad, inicialmente, en P.1 se exploraron mecanismos que permitían al personal médico definir cómo debe evolucionar la supervisión de los pacientes (es decir, las aplicaciones) ante cambios en su estado de salud. Para ello, se introdujeron dos conceptos: 1) concepto de evento, que permitía identificar situaciones relevantes ante las que era necesario reaccionar; 2) concepto de acción, para definir qué tareas se deben llevar a cabo en respuesta a dichos eventos. Tareas que se aplicaban sobre aplicaciones completas. Es importante destacar que tanto los eventos como las acciones tenían que ser registrados en el SR, de forma que se separaba la lógica de la aplicación de la lógica de adaptación. Desde el punto de vista de la gestión de aplicaciones, se dotó a la plataforma de un agente para la supervisión de los eventos (*Event Manager* en la Figura 1), responsable de recibir los avisos de eventos y del control de las acciones asociadas. Concretamente, se permitía crear

aplicaciones nuevas (iniciar la supervisión de nuevas variables biomédicas), parar aplicaciones existentes (dejar de controlar ciertas variables) o modificar los parámetros de ejecución de determinadas aplicaciones. De esta forma, se conseguía también una buena gestión de los recursos del sistema, que sólo se asignaban cuando era necesario. Eran los componentes de aplicación los encargados de la detección de situaciones relevantes (como resultado de la ejecución de su funcionalidad) y de su comunicación al Event Manager mediante invocación de métodos de su API.

Este proceso de adaptación conducido por la propia aplicación se generalizó en P.5 donde se estableció que la detección de eventos es una tarea propia de los agentes de aplicación, agentes de dominio encargados de capturar y procesar información del entorno, abstrayendo así a la plataforma de las particularidades del contexto. También se establecieron dos acciones genéricas, comunes a todos los campos de aplicación: crear y parar aplicaciones. De esta forma, la especificación de eventos y acciones propios del dominio se delega en la definición del meta-modelo de dominio. Finalmente, también se generalizó la comunicación entre el agente encargado de la gestión de eventos (*Event Agent* en la Figura 3), los agentes de dominio y el SRA. En concreto, los agentes de dominio avisan al agente de supervisión *Event Agent* de la ocurrencia de un evento invocando a métodos de su interfaz *IEvent* (ver Figura 10). Este agente busca en el SR, a través del API del SRA, las acciones asociadas a dicho evento y, finalmente, lanza y supervisa la ejecución de dichas acciones, a través de la interfaz *IAction*, que los agentes de dominio deben implementar con tareas propias de su ámbito.

Con respecto a la auto-recuperación, este proceso comprende dos tareas: la detección del fallo y la recuperación de la ejecución de la aplicación, incluso para el caso de aplicaciones con estado (aquellas cuyo estado de ejecución actual depende de los anteriores). Todo ello, de forma transparente para la aplicación. A lo largo de los diferentes trabajos publicados durante el desarrollo de la presente tesis doctoral se han realizado diferentes propuestas en torno a este proceso.

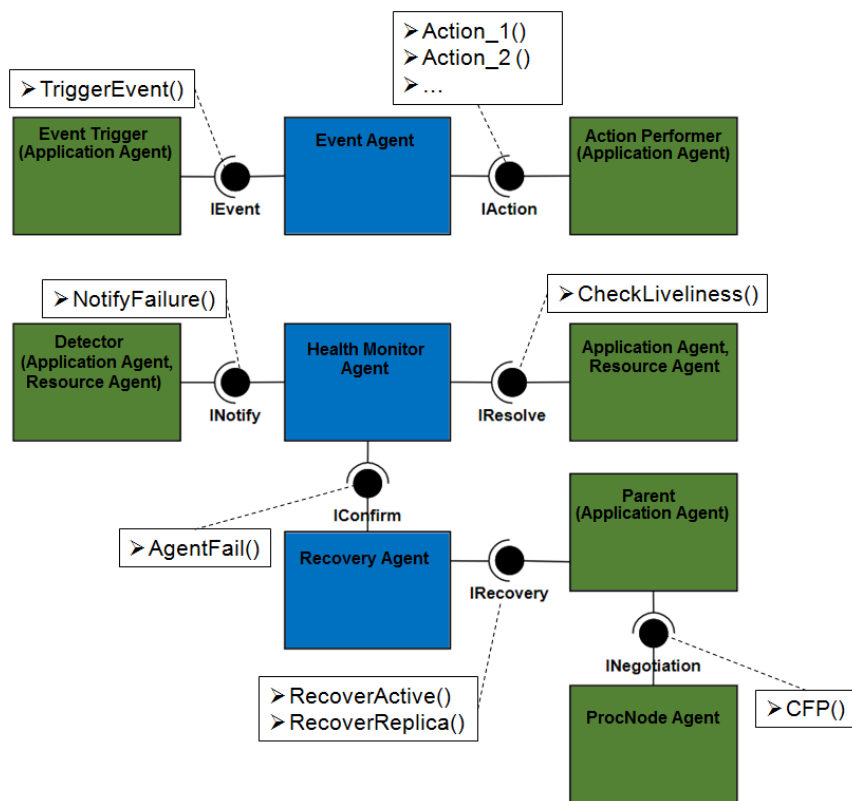


Figura 10: Interfaces definidos para las interacciones entre Agentes de Supervisión de Sistema (en color azul) y Agentes de dominio de Aplicación y Recurso (en color verde), en relación con los requisitos de flexibilidad: auto-adaptabilidad (R5) y auto-recuperación (R7).

Inicialmente, en P.1 se propuso una solución simple basada en la naturaleza móvil de los agentes, que también se emplea en P.3. La detección del fallo se fundamenta en la capacidad de JADE de avisar sobre la no entrega de mensajes ACL. Para el dominio de eHC también se propuso un mecanismo de detección de inactividad en agentes periódicos. Todos los avisos de agentes en fallo llegan a un gestor de la aplicación llamado *Application Manager* (ver Figura 1) quien inicia una tarea de recuperación por cada fallo, estableciendo un proceso de negociación entre todos los nodos de procesamiento disponibles (para ser más exactos, entre sus correspondientes agentes *Node Agent*). Se puede seleccionar así el nodo más adecuado y se crea en él una nueva

instancia del agente en fallo. El estado de la aplicación, es decir, el estado de los agentes que la componen, se guarda de forma centralizada en el SR. Por lo tanto, los agentes estarán dotados de los mecanismos necesarios para actualizar su estado en el SR tras cada ejecución.

Aunque esta solución asegura la recuperación de la ejecución de la aplicación, las decisiones tomadas pueden no ser las más adecuadas ya que los agentes *Application Manager* no disponen de visión global del estado del sistema (sólo conocen el estado de la aplicación que supervisan). Esta limitación se resuelve en la propuesta realizada en P.2 y P.4, que introduce dos novedades. Por un lado, la gestión de réplicas que permite liberar al SR del almacenamiento del estado, de forma que éste se transfiere de la instancia activa a sus réplicas. Así, se agiliza también el proceso de recuperación, ya que en caso de fallo de la instancia activa cualquiera de las réplicas puede ocupar su lugar. Por otro lado, la implementación de Agentes de Supervisión de Sistema para la trazabilidad de la QoS, que disponen de visión global del sistema lo que les permite realizar operaciones de diagnóstico y toma de decisiones más específicas. Es importante destacar que la propuesta realizada en P.2 y P.4 sería válida para cualquier QoS, siendo la disponibilidad de las aplicaciones un caso particular.

En esta propuesta, la detección del fallo es tarea de las réplicas de las instancias activas. Los avisos de fallos son recibidos por un agente de monitorización (perteneciente al grupo de agentes llamado *QoS Manager* en Figura 2) encargado de la confirmación del fallo y de evitar falsos positivos. Por su parte, un agente de diagnóstico y decisión (también perteneciente al grupo de agentes *QoS Manager* en Figura 2) toma las decisiones de recuperación oportunas, tras analizar el estado global del sistema. Nótese que las acciones de recuperación son propias del dominio y de la QoS monitorizada, por lo tanto, no genéricas. Por ejemplo, en el ámbito de FMS descrito en P.2 y P.4, y para el aseguramiento de la disponibilidad del servicio como QoS, dichas acciones de recuperación están enfocadas a lograr cambiar de una configuración de

controladores a otra diferente, de la manera más rápida y eficiente posible, sin interrumpir su funcionamiento normal.

Partiendo de la experiencia adquirida en estos trabajos previos y siguiendo la filosofía de toma de decisiones descentralizada con supervisión a nivel de sistema, en P.5 se generaliza el proceso de auto-recuperación basado en réplicas descrito en P.2 y P.4, con el objetivo de asegurar la disponibilidad de las aplicaciones en todo momento, incluyendo aplicaciones con estado y el caso de caída de nodos de procesamiento. Por un lado, la detección del fallo es distribuida y la llevan a cabo los agentes de dominio (tanto de aplicación como de recurso), haciendo uso de la potencialidad de JADE antes comentada. Por otro lado, dos Agentes de Supervisión de Sistema centralizan la verificación y recuperación del fallo (*Health Monitor Agent – HMA – y Recovery Agent – ReA –* en Figura 3). En P.5 se define una interfaz común a todos los dominios para estos Agentes de Supervisión de Sistema, mostrada en la Figura 10. En base a esta API, MAS-RECON también establece la secuencia de mensajes que deben intercambiar los diferentes agentes que intervienen en el proceso de auto-recuperación, desde la detección del fallo hasta su recuperación, independientemente del dominio. La Figura 11 muestra esta secuencia.

Los agentes de dominio avisan al HMA de la detección de un fallo mediante su interfaz *INotify (Detect* en Figura 11). El HMA hace uso de la interfaz *IResolve* implementada por los agentes de dominio para comprobar que se trata de un fallo real (*Resolve & Confirm* en Figura 11). Después, analiza si el fallo ha sido previamente notificado, descartándolo, o si se trata de un nuevo fallo, en cuyo caso avisa al agente ReA a través de su interfaz *IConfirm*. Finalmente, el agente ReA supervisa todo el proceso de recuperación que depende de si el agente en fallo es la instancia activa o una de las réplicas (*Recover* en Figura 11). Como se ha comentado anteriormente, al ser las acciones de recuperación propias del dominio, los agentes de dominio deben implementar la interfaz *IRecovery* para poder recibir órdenes del agente ReA.

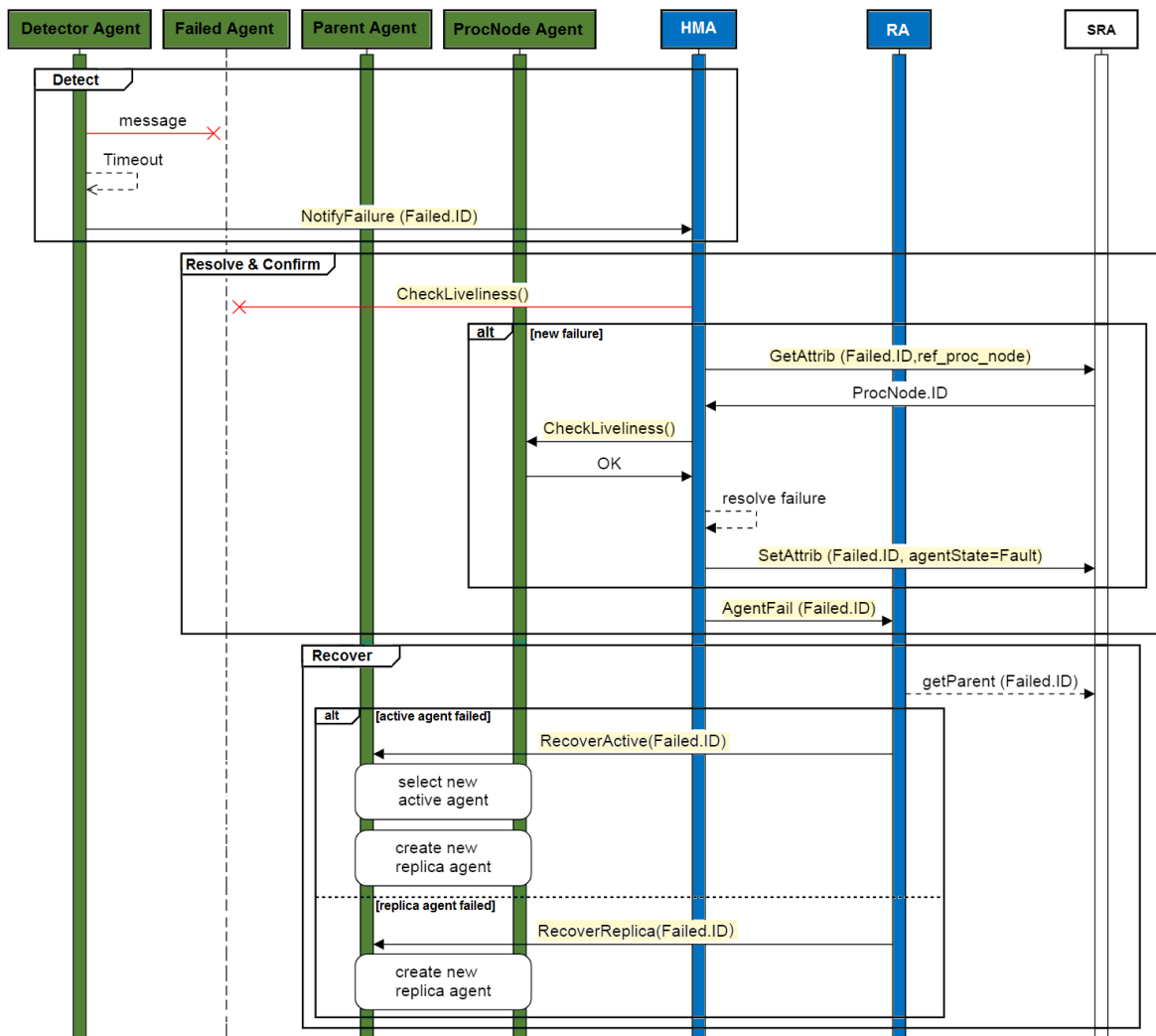


Figura 11: Diagrama de secuencia de los mensajes intercambiados durante el proceso de auto-recuperación: desde la detección del fallo hasta su recuperación.

1.4.4 Personalización y Extensión

En los apartados anteriores se ha descrito la propuesta de la plataforma genérica MAS-RECON, común a todos los dominios. Sin embargo, cuando se trabaja en un ámbito concreto es necesario personalizar, y a veces extender, MAS-RECON para obtener una plataforma que cubra las particularidades de dicho dominio. Una de las principales

aportaciones de P.5 es la metodología para implementar esta personalización y extensión (objetivo O.3.3).

El primer mecanismo para la personalización de MAS-RECON ya se ha introducido anteriormente y consiste en la definición del meta-modelo de dominio que incluye el concepto de aplicación. Así, el modelo del SR de cada dominio comprende tanto los recursos de dicho campo como la estructura de las aplicaciones, recogiendo sus características relevantes desde el punto de vista de su gestión y las relaciones entre todos ellos. Nótese que los eventos concretos a los que atender, así como las acciones a ejecutar como respuesta, también estarán recogidos en dicho modelo. En P.5 se detalla el proceso a seguir para la particularización del SR al dominio eHC.

El segundo mecanismo para la personalización de MAS-RECON está relacionado con el desarrollo de los agentes de dominio. En P.1 y P.2 se desarrollaron agentes de aplicación y/o de recurso para los dominios eHC y FMS, respectivamente, pudiendo identificarse aquellos comportamientos de los agentes comunes y específicos. Todo esto permitió que en P.5 se propusiera el esqueleto del código de los agentes de recurso y de aplicación que MAS-RECON proporciona. En ambos casos se trata de una implementación de las máquinas de estado (*Finite State Machine*, FSM) que definen el comportamiento genérico de dichos agentes y que se muestran en la Figura 12. Por lo tanto, el desarrollo de los agentes de dominio consiste en extender cada uno de los estados de la FSM con código que implemente las interacciones necesarias entre los diferentes agentes del sistema, para cumplir con las necesidades del dominio. Es importante recordar que, para algunas de estas interacciones, MAS-RECON ya establece algunas interfaces mínimas (ver Figura 8 y Figura 10) o incluso secuencia de mensajes (ver Figura 11). Es más, para algunas tareas comunes a todos los dominios, tales como la detección de fallo por mensaje no entregado, la recuperación de réplicas o el lanzamiento de procesos de negociación, MAS-RECON también proporciona el código asociado.

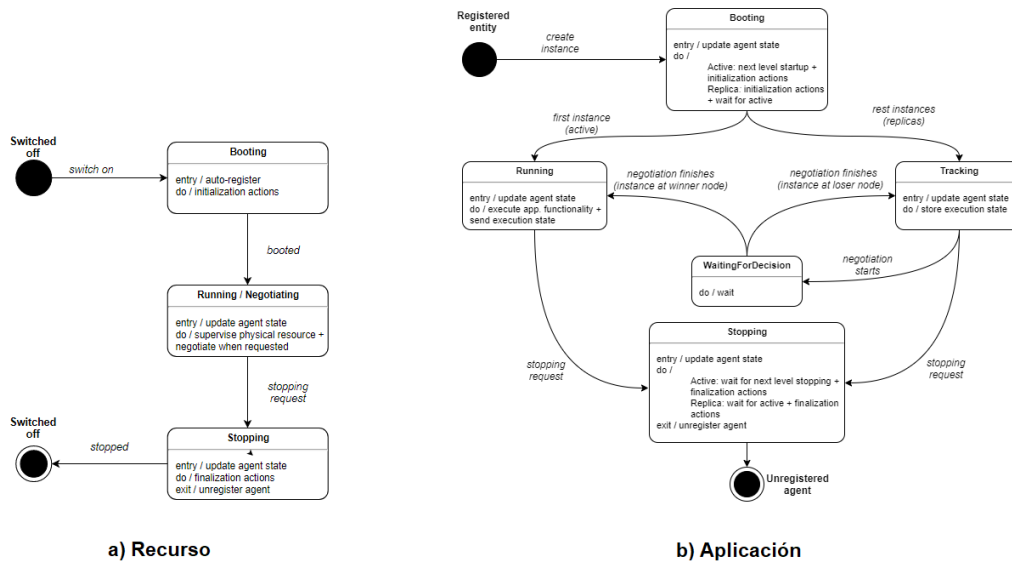


Figura 12: Diagramas de estados (FSM) correspondiente al comportamiento genérico de los agentes de domino: a) de recurso y b) de aplicación.

De esa manera, además de los nodos de procesamiento comunes a todos los dominios y proporcionados por MAS-RECON, es posible definir agentes de recurso del dominio para los cuales se pueden incluir acciones de inicialización y finalización concretas (estados *Booting* y *Stopping*, respectivamente). Además, se pueden establecer diferentes procesos de negociación definiendo criterios particulares y acciones concretas a realizar por los ganadores de dichas negociaciones (estado *Negotiating*). Finalmente, estos agentes propios de dominio también podrán implementar tareas específicas de supervisión de sus recursos (estado *Running*). Por ejemplo, en P.3 se extendió la funcionalidad de los nodos de procesamiento para que pudieran detectar e informar acerca de situaciones de sobreutilización e infrautilización de sus recursos. En cambio, en P.2 y P.4 fue necesario incorporar agentes encargados de la supervisión de recursos específicos de dominio, como los componentes mecatrónicos o las estaciones.

En lo que se refiere a los agentes de aplicación, en los estados *Booting* y *Stopping* también se pueden incluir acciones de inicialización y finalización, respectivamente. Además, en el estado de *Booting* también se debe implementar la fase 2 del proceso de

arranque antes descrito, la fase dependiente del dominio. En los estados *Running* y *Tracking* se incluye código relativo a la funcionalidad de la aplicación, así como el relativo a la supervisión específica de sus entidades. Por lo tanto, son estados centrados en la operación normal de las aplicaciones y en hacer frente a sus necesidades de flexibilidad. Para ello, en estos estados se deben implementar, como mínimo, las interfaces de la Figura 10 establecidas para los agentes de dominio. Por ejemplo, se pueden personalizar mecanismos de detección de fallo, como en P.1 donde se usa un tiempo límite para los componentes periódicos.

En P.5 se detalla el proceso de personalización de agentes de dominio a partir de las FSMs presentadas en la Figura 12, con el objetivo de lograr aquellos agentes de dominio que en P.1 se desarrollaron de forma ad-hoc para eHC. En P.4 se da un paso más y se demuestra cómo esta personalización se puede automatizar haciendo uso de la ingeniería conducida por modelos (*Model Driven Engineering*, MDE), aplicando transformaciones modelo-a-modelo (*Model-to-Model*, M2M) y modelo-a-texto (*Model-to-Text*, M2T) sobre modelos de información que recogen el comportamiento esperado de las diferentes entidades del dominio.

El último de los mecanismos de personalización propuesto por MAS-RECON se refiere a los Agentes de Supervisión de Sistema, permitiendo atender el cumplimiento de requisitos específicos de dominio. Por un lado, es posible añadir interfaces nuevos a los agentes de supervisión proporcionados en MAS-RECON, tal y como se hace en P.4 para supervisar diferentes parámetros de QoS: disponibilidad de la lógica de control y balanceo de carga. Para cada QoS se definen diferentes algoritmos de diagnóstico y decisión. Por otro lado, es posible extender MAS-RECON mediante la incorporación de nuevos Agentes de Supervisión de Sistema (objetivo O.3.4), siendo también necesario incluir el código para interactuar con estos nuevos agentes en las implementaciones de los agentes de dominio. Así, para hacer frente a las necesidades de QoS flexible (aplicaciones que aceptan cierta degradación de su calidad), como ocurre en aplicaciones multimedia, se pueden incorporar mecanismos que permitan adecuar la

calidad de las aplicaciones a los recursos disponibles y también incorporar algoritmos avanzados de control de admisión.

1.4.5 Análisis de Rendimiento

Una vez propuesta la plataforma genérica y habiéndola implementado sobre JADE, se procedió a realizar un análisis de su rendimiento en diferentes aspectos: despliegue de aplicaciones y recuperación ante fallos.

Por un lado, se analizó el rendimiento de MAS-RECON con respecto al despliegue de aplicaciones, haciendo uso de una aplicación muy simple formada por componentes (entidades de aplicación) conectados en cadena. En primer lugar, se midieron los tiempos de despliegue para diferentes cargas de trabajo sobre un sistema formado por un número fijo de 20 nodos y con una carga máxima de 600 componentes. En segundo lugar, los tiempos de despliegue se analizaron manteniendo el sistema en una situación de carga máxima y variando el número de nodos entre los que repartirla.

La Figura 13 recoge las medidas del primer test, donde se muestran los tiempos medios de (a) planificación, (b) despliegue y (c) arranque para las siguientes cargas de trabajo: del 10% (60 componentes); 25% (150 componentes); 50% (300 componentes); 75% (450 componentes); y 100% (600 componentes). Para poder disponer de una medida relativa, estas mismas pruebas se realizaron también sobre Kubernetes, que, tal y como se ha indicado anteriormente, se considera la herramienta más extendida para la orquestación de aplicaciones basadas en microservicios e implementadas mediante contenedores. Como se puede observar en la figura, Kubernetes es más rápido en la planificación (ver Figura 13.a), y también en el arranque de componentes (ver Figura 13.c), en este caso sobre todo para cargas de trabajo bajas, ya que para cargas altas los tiempos convergen. Sin embargo, MAS-RECON es más rápido en la creación y despliegue de componentes (ver Figura 13.b).

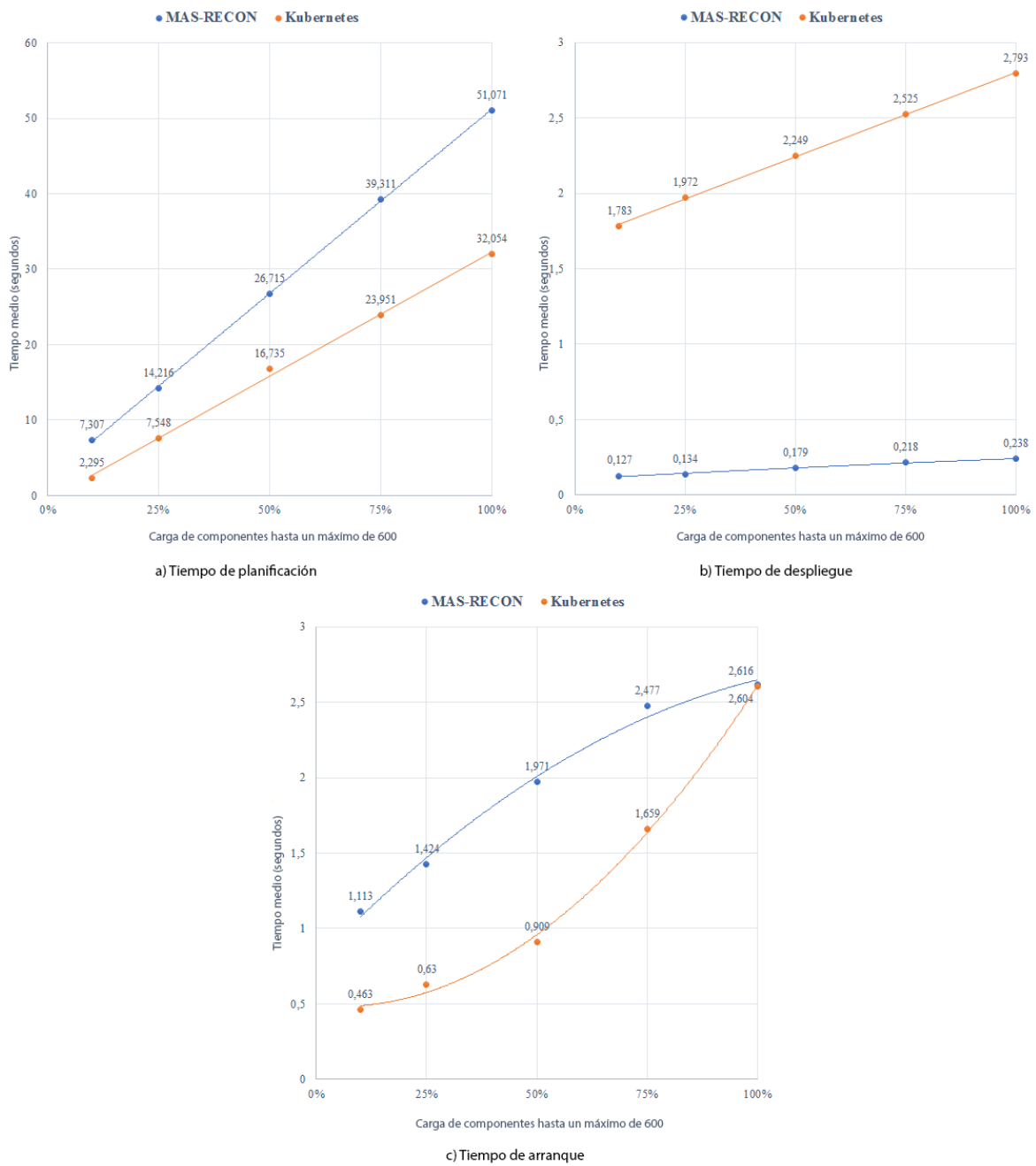


Figura 13: Tiempos de planificación (a), despliegue (b) y arranque (c) de componentes en MAS-RECON (color azul), en comparación con la plataforma de orquestación Kubernetes (color naranja), en función de la carga de trabajo del sistema.

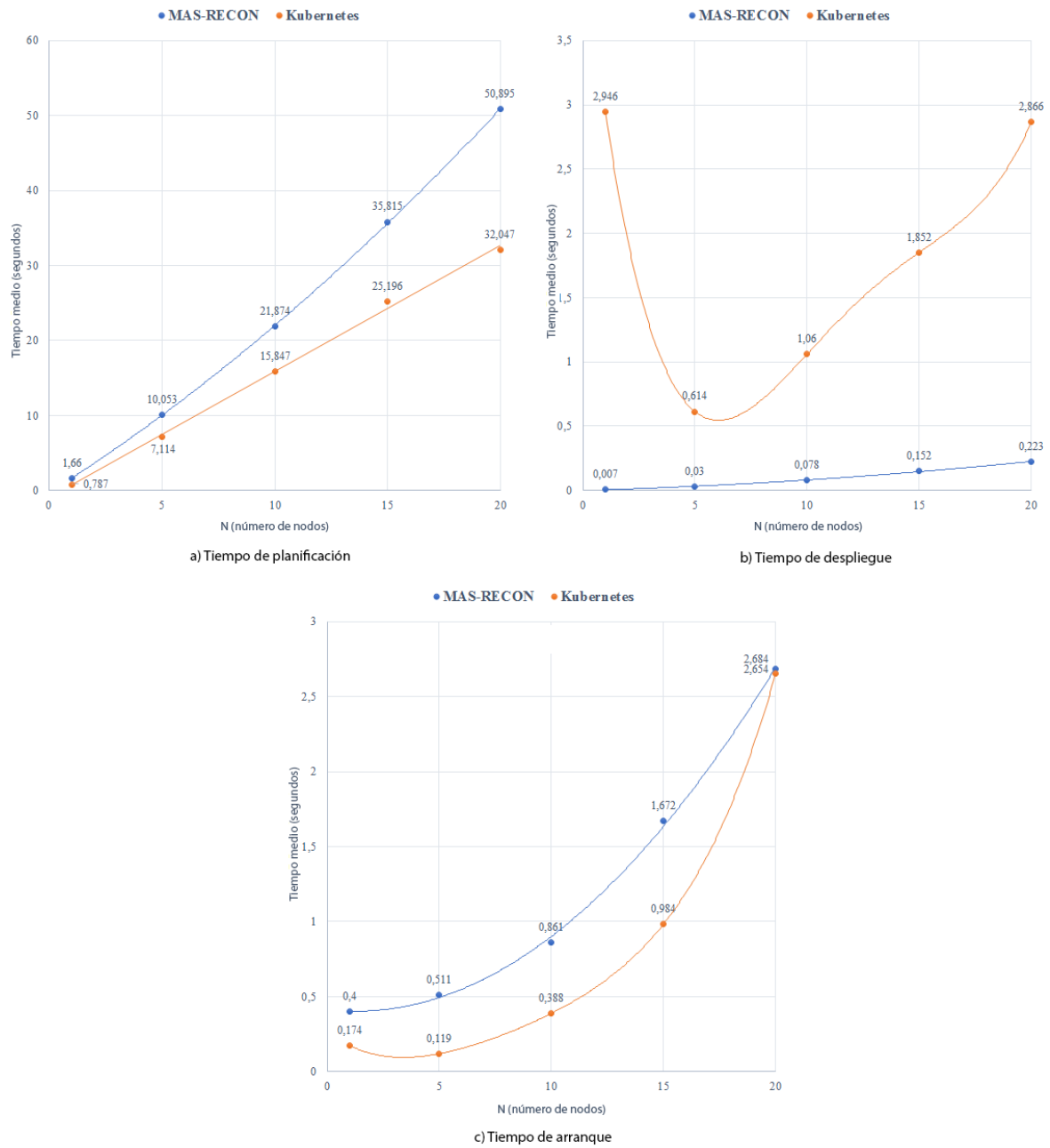


Figura 14: Tiempos de planificación (a), despliegue (b) y arranque (c) de componentes en MAS-RECON (color azul), en comparación con la plataforma de orquestación Kubernetes (color naranja), en función del número de nodos del sistema.

La Figura 14 recoge las medidas del segundo test y muestra los tiempos medios de (a) planificación, (b) despliegue y (c) arranque para el siguiente número de nodos (N):

N=1, N=5, N=10, N=15, N=20. Para cada prueba se mantuvo una carga de trabajo del 100%, equivalente a $30 \cdot N$ componentes. El análisis de los resultados obtenidos es similar al del primer test en lo que se refiere a planificación y despliegue. Sin embargo, en este caso se observa que los tiempos de arranque son mejores en Kubernetes para sistemas pequeños, y que a medida que el número de nodos aumenta esta diferencia entre MAS-RECON y Kubernetes disminuye, pudiendo deducir que para sistemas grandes MAS-RECON será mejor.

De estos dos test se puede concluir que los tiempos de MAS-RECON pueden ser ligeramente peores a los de otras plataformas. Sin embargo, lejos de ser una limitación este aumento en los tiempos medidos se debe a la gran capacidad de personalización y adaptación al dominio que ofrece MAS-RECON, y que sobrecarga ligeramente al sistema. De hecho, el algoritmo de arranque implementado para MAS-RECON en las pruebas permitía un arranque ordenado y sincronizado de los diferentes componentes concatenados (del último al primero), evitando incoherencias en la ejecución de la aplicación. Este arranque sincronizado es imposible de conseguir en Kubernetes sin personalizar sus mecanismos de despliegue.

Con respecto a los tiempos de recuperación ante fallos de MAS-RECON, se realizaron pruebas en un sistema formado por un número fijo de nodos (6 nodos) entre los que se desplegaron aplicaciones de diferentes tamaños (N componentes). El despliegue inicial de estas aplicaciones, con un factor de réplica 0 (es decir, sin réplicas) aseguraba el balanceo de carga entre los diferentes nodos, por lo que la carga de cada uno de ellos era de $N/6$ componentes. Se realizaron pruebas en las que se provocaba el fallo de uno de los 6 nodos para los siguientes tamaños de aplicación: 60 componentes (recuperación de 10 componentes), 150 componentes (recuperación de 25 componentes), 300 componentes (recuperación de 50 componentes), 450 componentes (recuperación de 75 componentes) y 600 componentes (recuperación de 100 componentes). En cada prueba se midieron 3 tiempos: 1) tiempo de reacción: tiempo transcurrido desde el fallo hasta que la plataforma empieza a responder. Es

decir, hasta que el HMA recibe la notificación del fallo; 2) tiempo de restauración: tiempo transcurrido desde la detección del fallo hasta que se recupera el primero de los componentes afectados; y 3) tiempo de recuperación: tiempo transcurrido desde la detección del fallo hasta que se restaura la totalidad de la aplicación.

La Tabla 4 muestra los tiempos obtenidos para las diferentes pruebas realizadas. Como se puede observar, el tiempo de reacción no sufre variaciones ya que la detección del fallo es una tarea independiente de la carga del sistema. Algo similar ocurre con los tiempos de restauración, ya que se miden con respecto al primer componente recuperado, independientemente del número total de componentes afectados. Sin embargo, como era de esperar, el tiempo de recuperación sí que aumenta con el número total de componentes afectados, pero el ratio entre este tiempo y el número de componentes se mantiene constante (1 segundo aproximadamente).

Tabla 4: Tiempos de recuperación ante fallos de aplicaciones de diferente tamaño (N). El fallo únicamente afecta a uno de los nodos (i.e., pérdida de N/6 componentes)

	N=60	N=150	N=300	N=450	N=600
Reacción	3,14s	3,14s	3,14s	3,54s	3,81s
Restauración	5,28s	5,56s	5,33s	4,35s	5,97s
Recuperación	8,38s	18,62s	52,87s	85,81s	96,16s

1.5 Referencias

- Agirre, A., Parra, J., Armentia, A., Estévez, E., Marcos, M., 2016. QoS Aware Middleware Support for Dynamically Reconfigurable Component Based IoT Applications. *Int. J. Distrib. Sens. Netw.* 2016, 1–17. <https://doi.org/10.1155/2016/2702789>
- Albassam, E., Porter, J., Gomaa, H., Menasce, D.A., 2017. DARE: A Distributed Adaptation and Failure Recovery Framework for Software Systems, in: *2017 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, Columbus, OH, USA, pp. 203–208. <https://doi.org/10.1109/ICAC.2017.12>

- Al-Jaroodi, J., Mohamed, N., 2012. Service-oriented middleware: A survey. *J. Netw. Comput. Appl.* 35, 211–220. <https://doi.org/10.1016/j.jnca.2011.07.013>
- Arcaini, P., Riccobene, E., Scandurra, P., 2015. Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation, in: 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. IEEE, Florence, Italy, pp. 13–23. <https://doi.org/10.1109/SEAMS.2015.10>
- Argente, E., Botti, V., Carrascosa, C., Giret, A., Julian, V., Rebollo, M., 2011. An abstract architecture for virtual organizations: The THOMAS approach. *Knowl. Inf. Syst.* 29, 379–403. <https://doi.org/10.1007/s10115-010-0349-1>
- Ayaz, M., Ammad-Uddin, M., Sharif, Z., Mansour, A., Aggoune, E.-H.M., 2019. Internet-of-Things (IoT)-Based Smart Agriculture: Toward Making the Fields Talk. *IEEE Access* 7, 129551–129583. <https://doi.org/10.1109/ACCESS.2019.2932609>
- Baek, J., Alhindi, T.J., Jeong, Y.-S., Jeong, M.K., Seo, S., Kang, J., Heo, Y., 2021. Intelligent Multi-Sensor Detection System for Monitoring Indoor Building Fires. *IEEE Sens. J.* 21, 27982–27992. <https://doi.org/10.1109/JSEN.2021.3124266>
- Baker, S.B., Xiang, W., Atkinson, I., 2017. Internet of Things for Smart Healthcare: Technologies, Challenges, and Opportunities. *IEEE Access* 5, 26521–26544. <https://doi.org/10.1109/ACCESS.2017.2775180>
- Bellifemine, F., Caire, G., Poggi, A., Rimassa, G., 2008. JADE: A software framework for developing multi-agent applications. Lessons learned. *Inf. Softw. Technol.* 50, 10–21. <https://doi.org/10.1016/j.infsof.2007.10.008>
- Boudaa, B., Hammoudi, S., Benslimane, S.M., 2018. Towards an Extensible Context Model for Mobile User in Smart Cities, in: Computational Intelligence and Its Applications. CIIA 2018. IFIP Advances in Information and Communication Technology, IFIP Advances in Information and Communication Technology. Springer International Publishing, Springer, Cham, pp. 498–508. https://doi.org/10.1007/978-3-319-89743-1_43
- Boyes, H., Hallaq, B., Cunningham, J., Watson, T., 2018. The industrial internet of things (IIoT): An analysis framework. *Comput. Ind.* 101, 1–12. <https://doi.org/10.1016/j.compind.2018.04.015>
- Chen, B., Wan, J., Shu, L., Li, P., Mukherjee, M., Yin, B., 2018. Smart Factory of Industry 4.0: Key Technologies, Application Case, and Challenges. *IEEE Access* 6, 6505–6519. <https://doi.org/10.1109/ACCESS.2017.2783682>
- Čolaković, A., Hadžialić, M., 2018. Internet of Things (IoT): A review of enabling technologies, challenges, and open research issues. *Comput. Netw.* 144, 17–39. <https://doi.org/10.1016/j.comnet.2018.07.017>
- Coutinho, L.R., Brandão, A.A.F., Boissier, O., Sichman, J.S., 2019. Towards Agent Organizations Interoperability: A Model Driven Engineering Approach. *Appl. Sci.* 9, 1–38. <https://doi.org/10.3390/app9122420>
- Farooq, M.S., Riaz, S., Abid, A., Abid, K., Naeem, M.A., 2019. A Survey on the Role of IoT in Agriculture for the Implementation of Smart Farming. *IEEE Access* 7, 156237–156271. <https://doi.org/10.1109/ACCESS.2019.2949703>
- Foundation for Intelligent Physical Agents, 2005. Standard FIPA specifications [WWW Document]. URL <http://www.fipa.org/specifications/> (accessed 9.8.22).

- Funk, C., Ehm, C., Linnhoff-Popien, C., Kuhmunch, C., 2007. Support of Stateful Services in Pervasive Environments, in: Fifth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PerComW'07). IEEE, White Plains, NY, USA, pp. 483–488. <https://doi.org/10.1109/PERCOMW.2007.110>
- Galster, M., Weyns, D., Tofan, D., Michalik, B., Avgeriou, P., 2014. Variability in Software Systems—A Systematic Literature Review. *IEEE Trans. Softw. Eng.* 40, 282–306. <https://doi.org/10.1109/TSE.2013.56>
- Garcia Valls, M., Lopez, I.R., Villar, L.F., 2013. iLAND: An Enhanced Middleware for Real-Time Reconfiguration of Service Oriented Distributed Real-Time Systems. *IEEE Trans. Ind. Inform.* 9, 228–236. <https://doi.org/10.1109/TII.2012.2198662>
- García-Magariño, I., Gutiérrez, C., 2013. Agent-oriented modeling and development of a system for crisis management. *Expert Syst. Appl.* 40, 6580–6592. <https://doi.org/10.1016/j.eswa.2013.06.012>
- Garruzzo, S., Rosaci, D., Sarne, G.M.L., 2007. MASHA-EL: A Multi-Agent System for Supporting Adaptive E-Learning, in: 19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2007). IEEE, Patras, Greece, pp. 103–110. <https://doi.org/10.1109/ICTAI.2007.83>
- Gómez-Sanz, J.J., Fuentes-Fernández, R., 2015. Understanding Agent-Oriented Software Engineering methodologies. *Knowl. Eng. Rev.* 30, 375–393. <https://doi.org/10.1017/S0269888915000053>
- Guerraoui, R., Schiper, A., 1997. Software-Based Replication for Fault Tolerance. *Computer* 30, 68–74. <https://doi.org/10.1109/2.585156>
- Gui, N., De Florio, V., Sun, H., Blondia, C., 2011. Toward architecture-based context-aware deployment and adaptation. *J. Syst. Softw.* 84, 185–197. <https://doi.org/10.1016/j.jss.2010.09.017>
- Hallsteinsen, S., Geihs, K., Paspallis, N., Eliassen, F., Horn, G., Lorenzo, J., Mamelli, A., Papadopoulos, G.A., 2012. A development framework and methodology for self-adapting applications in ubiquitous computing environments. *J. Syst. Softw.* 85, 2840–2859. <https://doi.org/10.1016/j.jss.2012.07.052>
- Hassija, V., Chamola, V., Saxena, V., Jain, D., Goyal, P., Sikdar, B., 2019. A Survey on IoT Security: Application Areas, Security Threats, and Solution Architectures. *IEEE Access* 7, 82721–82743. <https://doi.org/10.1109/ACCESS.2019.2924045>
- He, X., Tu, Z., Xu, X., Wang, Z., 2021. Programming framework and infrastructure for self-adaptation and optimized evolution method for microservice systems in cloud-edge environments. *Future Gener. Comput. Syst.* 118, 263–281. <https://doi.org/10.1016/j.future.2021.01.008>
- Hsu, T.-Y., Nieh, C.P., 2020. On-Site Earthquake Early Warning Using Smartphones. *Sensors* 20, 2928. <https://doi.org/10.3390/s20102928>
- Huan, W., Hidenori, N., 2012. Failure Detection in P2P-Grid Environments, in: 32nd International Conference on Distributed Computing Systems Workshops. IEEE, Macau, China, pp. 369–374. <https://doi.org/10.1109/ICDCSW.2012.18>
- Hussein, M., Han, J., Colman, A., 2011. An Approach to Model-Based Development of Context-Aware Adaptive Systems, in: 2011 35th IEEE Annual Computer

- Software and Applications Conference. IEEE, Munich, Germany, pp. 205–214. <https://doi.org/10.1109/COMPSAC.2011.34>
- Iqbal, M.W., Ch, N.A., Shahzad, S.K., Naqvi, M.R., Khan, B.A., Ali, Z., 2021. User Context Ontology for Adaptive Mobile-Phone Interfaces. *IEEE Access* 9, 96751–96762. <https://doi.org/10.1109/ACCESS.2021.3095300>
- Isern, D., Sánchez, D., Moreno, A., 2011. Organizational structures supported by agent-oriented methodologies. *J. Syst. Softw.* 84, 169–184. <https://doi.org/10.1016/j.jss.2010.09.005>
- Islam, S.M.R., Kwak, D., Kabir, M.H., Hossain, M., Kwak, K.-S., 2015. The Internet of Things for Health Care: A Comprehensive Survey. *IEEE Access* 3, 678–708. <https://doi.org/10.1109/ACCESS.2015.2437951>
- James Lewis, Martin Fowler, 2014. Microservices [WWW Document]. URL <https://martinfowler.com/articles/microservices.html> (accessed 7.20.21).
- Khabou, N., Rodriguez, I.B., Gharbi, G., Jmaiel, M., 2014. A Threshold based Context Change Detection in Pervasive Environments: Application to a Smart Campus, in: 5th International Conference on Ambient Systems, Networks and Technologies (ANT-2014). Hasselt, Belgium, pp. 461–468. <https://doi.org/10.1016/j.procs.2014.05.448>
- Khan, M.U., Reichle, R., Geihs, K., 2008. Architectural Constraints in the Model-Driven Development of Self-Adaptive Applications. *IEEE Distrib. Syst. Online* 9, 1–10. <https://doi.org/10.1109/MDSO.2008.19>
- Krupitzer, C., Roth, F.M., VanSyckel, S., Schiele, G., Becker, C., 2015. A survey on engineering approaches for self-adaptive systems. *Pervasive Mob. Comput.* 17, 184–206. <https://doi.org/10.1016/j.pmcj.2014.09.009>
- Kubernetes, 2022. Kubernetes [WWW Document]. URL <https://kubernetes.io/> (accessed 3.10.22).
- Li, X., Eckert, M., Martinez, J.-F., Rubio, G., 2015. Context Aware Middleware Architectures: Survey and Challenges. *Sensors* 15, 20570–20607. <https://doi.org/10.3390/s150820570>
- Michael Wooldridge, 2009. *An Introduction to MultiAgent Systems*, 2nd ed. Wiley Publishing, Hoboken, NJ, USA.
- Mohamed, M.A., Kardas, G., Challenger, M., 2021. Model-Driven Engineering Tools and Languages for Cyber-Physical Systems—A Systematic Literature Review. *IEEE Access* 9, 48605–48630. <https://doi.org/10.1109/ACCESS.2021.3068358>
- Perera, C., Liu, C.H., Jayawardena, S., Chen, M., 2014. A Survey on Internet of Things From Industrial Market Perspective. *IEEE Access* 2, 1660–1679. <https://doi.org/10.1109/ACCESS.2015.2389854>
- Psaier, H., Dustdar, S., 2011. A survey on self-healing systems: approaches and systems. *Computing* 91, 43–73. <https://doi.org/10.1007/s00607-010-0107-y>
- Qi, Q., Tao, F., 2019. A Smart Manufacturing Service System Based on Edge Computing, Fog Computing, and Cloud Computing. *IEEE Access* 7, 86769–86777. <https://doi.org/10.1109/ACCESS.2019.2923610>

- Ramírez, A., Moreno, N., Vallecillo, A., 2021. Rule-based preprocessing for data stream mining using complex event processing. *Expert Syst.* 38, e12762. <https://doi.org/10.1111/exsy.12762>
- Ray, P.P., Mukherjee, M., Shu, L., 2017. Internet of Things for Disaster Management: State-of-the-Art and Prospects. *IEEE Access* 5, 18818–18835. <https://doi.org/10.1109/ACCESS.2017.2752174>
- Rocha, A., Martins, A., Freire, J.C., Kamel Boulos, M.N., Vicente, M.E., Feld, R., van de Ven, P., Nelson, J., Bourke, A., ÓLaighin, G., Sdogati, C., Jobes, A., Narvaiza, L., Rodríguez-Moliner, A., 2013. Innovations in health care services: The CAALYX system. *Int. J. Med. Inf.* 82, e307–e320. <https://doi.org/10.1016/j.ijmedinf.2011.03.003>
- Rosaci, D., Sarné, G.M.L., 2006. MASHA: A multi-agent system handling user and device adaptivity of Web sites. *User Model. User-Adapt. Interact.* 16, 435–462. <https://doi.org/10.1007/s11257-006-9015-4>
- Ruiz, A., Juez, G., Schleiss, P., Weiss, G., 2015. A safe generic adaptation mechanism for smart cars, in: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE). IEEE, Gaithersbury, MD, USA, pp. 161–171. <https://doi.org/10.1109/ISSRE.2015.7381810>
- Shah, S.A., Seker, D.Z., Rathore, M.M., Hameed, S., Ben Yahia, S., Draheim, D., 2019. Towards Disaster Resilient Smart Cities: Can Internet of Things and Big Data Analytics Be the Game Changers? *IEEE Access* 7, 91885–91903. <https://doi.org/10.1109/ACCESS.2019.2928233>
- Vale, T., Crnkovic, I., de Almeida, E.S., Silveira Neto, P.A. da M., Cavalcanti, Y.C., Meira, S.R. de L., 2016. Twenty-eight years of component-based software engineering. *J. Syst. Softw.* 111, 128–148. <https://doi.org/10.1016/j.jss.2015.09.019>
- Vayghan, L.A., Saied, M.A., Toeroe, M., Khendek, F., 2021. A Kubernetes controller for managing the availability of elastic microservice based stateful applications. *J. Syst. Softw.* 175, 1–13. <https://doi.org/10.1016/j.jss.2021.110924>
- Villarrubia, G., Hernández, D., De Paz, J.F., Bajo, J., 2017. Combination of multi-agent systems and embedded hardware for the monitoring and analysis of diuresis. *Int. J. Distrib. Sens. Netw.* 13, 1–17. <https://doi.org/10.1177/1550147717722154>
- Wang, Y., Kadiyala, H., Rubin, J., 2021. Promises and challenges of microservices: an exploratory study. *Empir. Softw. Eng.* 26, 1–44. <https://doi.org/10.1007/s10664-020-09910-y>
- Xu, H., Yu, W., Griffith, D., Golmie, N., 2018. A Survey on Industrial Internet of Things: A Cyber-Physical Systems Perspective. *IEEE Access* 6, 78238–78259. <https://doi.org/10.1109/ACCESS.2018.2884906>
- Zato, C., Villarrubia, G., Sánchez, A., Bajo, J., Corchado, J.M., 2013. PANGEA: A New Platform for Developing Virtual Organizations of Agents. *Int. J. Artif. Intell.* 11, 93–102.

2 CONCLUSIONES

2.1 Conclusiones

En la presente tesis doctoral se ha propuesto la plataforma MAS-RECON para la gestión de la ejecución de las aplicaciones sensibles al contexto, que cubre tanto sus requisitos operacionales como los de flexibilidad. MAS-RECON es una plataforma genérica para todos los dominios que al mismo tiempo es personalizable y extensible a dominios concretos. Por lo tanto, MAS-RECON no sólo cubre las necesidades de gestión comunes a todos los campos, sino que también puede hacer frente a las necesidades específicas del dominio. A lo largo de este trabajo se han realizado propuestas en distintos dominios de aplicación (eHC, FMS y multimedia) permitieron identificar: 1) requisitos operacionales de la plataforma comunes a todos los dominios; 2) el concepto de aplicación, dependiente del dominio; y 3) requisitos de flexibilidad, comunes y específicos de dominio. En cada uno de los campos se propuso una solución ad-hoc para los problemas particulares, y de todos ellos emergió el resultado principal de este trabajo: una plataforma base que ofrece servicios básicos de gestión de aplicaciones, como arranque, parada y operación normal, y servicios de flexibilidad comunes a todos los dominios, como la auto-adaptabilidad y la auto-recuperación. Por otro lado, dando solución a la especificidad del campo, la plataforma MAS-RECON es capaz de personalizarse, fundamentalmente en base a la definición de un meta-modelo del dominio y otros mecanismos relacionados con el desarrollo del código de los agentes.

Se ha probado que el uso de tecnología de meta-modelado permite tanto la definición como la gestión genérica del estado del sistema, propio de cada campo. Siendo una tecnología clave para hacer frente a la variabilidad de dominio, ya que permite definir de forma genérica el concepto de aplicación, propio de cada campo. Así, MAS-RECON considera el concepto de la aplicación desde las primeras etapas de su diseño, entendida como un conjunto de entidades que se relacionan de una forma específica dentro de un dominio. Esto permite que MAS-RECON pueda ofrecer una gestión de la ejecución centrada en las propias aplicaciones y no en los módulos que las componen.

También se ha demostrado que la tecnología multi-agente es adecuada para llevar a cabo la idea de inteligencia distribuida con supervisión centralizada. Así, es posible hacer frente a sucesos inesperados como pueden ser cambios en el contexto o fallo de agentes y recursos. Concretamente, se ha introducido inteligencia dentro de las entidades de dominio, encargadas de la detección de estos sucesos inesperados, de la toma de decisiones distribuida y de la ejecución de tareas concretas de reacción y/o recuperación. Se propone el uso de agentes de sistema con acceso al estado global del sistema para la supervisión centralizada de los agentes de dominio. De esta forma se puede hacer frente a todos aquellos requisitos que afecten al conjunto de la aplicación, como es el caso de la auto-adaptabilidad y auto-recuperación. Como resultado, es posible separar la lógica de adaptación a cambios de contexto de la lógica de aplicación. También es posible el despliegue descentralizado de las aplicaciones y la recuperación rápida del servicio para aplicaciones con estado, incluso en el caso de fallo de nodo.

MAS-RECON proporciona la estructura que debe seguir el repositorio de sistema de cualquier dominio, el API de aquellos agentes de sistema comunes a todos los dominios y el esqueleto del código de los agentes de dominio, tanto de recurso como de aplicación, junto con una implementación básica de aquellos mecanismos e interfaces que estos agentes necesitan para su comunicación con los agentes de sistema.

Por otro lado, MAS-RECON también establece una metodología de personalización a dominio y extensión basada, fundamentalmente, en la definición del meta-modelo de dominio, que describe la estructura de las aplicaciones, y en el desarrollo de plantillas de agentes, que implementan las comunicaciones definidas entre las entidades del dominio y los agentes de sistema. De hecho, se puede decir que MAS-RECON va más allá que otras plataformas de gestión, siendo una especie de entorno de desarrollo que facilita la implementación de agentes mediante la definición de plantillas.

Se puede concluir, por lo tanto, que MAS-RECON junto con esta metodología de personalización permite desarrollar plataformas específicas de dominio que cumplen

con los requisitos de las aplicaciones sensibles al contexto, tanto los operacionales como los de flexibilidad, y tanto los comunes como los específicos de dominio.

Sin embargo, resulta importante destacar dos limitaciones de MAS-RECON. Por un lado, tal y como se ha observado en el análisis de rendimiento realizado, el hecho de que sea una plataforma genérica y personalizable hace que su rendimiento sea ligeramente peor que el de otras plataformas genéricas, pero cuya personalización resulta más laboriosa. Se debería, por lo tanto, evaluar para cada caso concreto si optar por MAS-RECON y su capacidad para hacer frente a requisitos específicos del dominio, u optar por otra alternativa menos personalizable pero óptima.

La otra limitación que presenta MAS-RECON está relacionada con el hecho de que no cubre el requisito de seguridad, ya que se ha considerado que podría extenderse mediante mecanismos existentes que cada usuario debería incorporar. Esto hace que MAS-RECON no sea una plataforma segura, aspecto que actualmente está cobrando una importancia vital.

2.2 Trabajo Futuro

La primera línea de trabajo está relacionada con la segunda limitación identificada en el apartado anterior y consiste en estudiar los mecanismos de seguridad existentes e incorporarlos en el núcleo genérico que proporciona MAS-RECON, analizando cómo ello afecta a su rendimiento. Así, MAS-RECON podría ser una plataforma genérica, personalizable, extensible y segura.

Finalmente, otra línea de trabajo futuro podría estar enfocada al desarrollo de plataformas genéricas y personalizables dentro de dominios concretos (meta-plataformas). En efecto, el esfuerzo necesario para particularizar MAS-RECON a un dominio concreto puede resultar un inconveniente, ya que integrar nuevos agentes

específicos requiere un conocimiento profundo de la plataforma. Así, tras analizar en detalle todas las necesidades de un campo concreto, y haciendo uso de la ingeniería conducida por modelos y las transformaciones de modelos, sería posible personalizar MAS-RECON para obtener una plataforma y unas plantillas de agentes de dominio que, a su vez, serían personalizables para un conjunto de aplicaciones concreto.

3 ANEXO: PUBLICACIONES

3.1 Flexibility Support for Homecare Applications Based on Models and Multi-Agent Technology

Armentia, A., Gangoiti, U., Priego, R., Estévez, E., y Marcos, M. (2015). Flexibility Support for Homecare Applications Based on Models and Multi-Agent Technology. *Sensors*, 15 (12), pp. 31939–31964.

DOI: <https://doi.org/10.3390/s151229899>.

JCR©2015: 2,033

Categoría: Instruments & Instrumentation

Cuartil: Q1 (12/56)

Article

Flexibility Support for Homecare Applications Based on Models and Multi-Agent Technology

Aintzane Armentia ^{1,*}, Unai Gangoiti ^{1,†}, Rafael Priego ^{1,†}, Elisabet Estévez ^{2,†} and Marga Marcos ^{1,†}

Received: 10 November 2015; Accepted: 13 December 2015; Published: 17 December 2015

Academic Editor: Vittorio M. N. Passaro

¹ Automatic Control & Systems Engineering Department, ETSI Bilbao, University of the Basque Country (UPV/EHU), 48013 Bilbao, Spain; unai.gangoiti@ehu.eus (U.G.); rafael.priego@ehu.eus (R.P.); marga.marcos@ehu.eus (M.M.)

² Electronic and Automation Engineering Department, University of Jaen (UJA), 23071 Jaén, Spain; eestevez@ujaen.es

* Correspondence: aintzane.armentia@ehu.eus; Tel.: +34-946-017-216; Fax: +34-946-014-187

† These authors contributed equally to this work.

Abstract: In developed countries, public health systems are under pressure due to the increasing percentage of population over 65. In this context, homecare based on ambient intelligence technology seems to be a suitable solution to allow elderly people to continue to enjoy the comforts of home and help optimize medical resources. Thus, current technological developments make it possible to build complex homecare applications that demand, among others, flexibility mechanisms for being able to evolve as context does (adaptability), as well as avoiding service disruptions in the case of node failure (availability). The solution proposed in this paper copes with these flexibility requirements through the whole life-cycle of the target applications: from design phase to runtime. The proposed domain modeling approach allows medical staff to design customized applications, taking into account the adaptability needs. It also guides software developers during system implementation. The application execution is managed by a multi-agent based middleware, making it possible to meet adaptation requirements, assuring at the same time the availability of the system even for stateful applications.

Keywords: AAL systems; homecare; adaptability; availability; stateful components; domain modeling; multi-agent systems

1. Introduction

Developed countries are suffering a demographic change as a result of the growing number of older people as well as an increase in longevity [1–3]. Elderly suffer from typical age-related diseases demanding expensive medical care that is pressuring public health systems. Governments, conscious of this problem, are funding research projects aiming at providing new ways of medical care [4–7]. Indeed, issues like extending healthy life expectancy, improving quality of life, and maintaining autonomy and independence, are part of the term “active ageing” that was adopted by the World Health Organization in the late 1990s [8]. In this context, home-based care solutions seem useful to provide personalized care, improve comfort, autonomy, confidence and safety of the residents, optimizing, at the same time, medical resources [2,9,10].

During the last years, Ambient Assisted Living (AAL) systems have emerged as an adequate technological support for elderly and disabled to enhance their quality of life avoiding social isolation [11–15]. AAL systems have been studied by several authors with different purposes, from energy efficiency or comfort optimization to dealing with safety or recognizing elderly activity [16–22], as well as for home care [11,14]. In the particular case of home care for elderly, smart homes are

equipped with sensors, actuators and other appliances, whereas patients are provided with medical sensors and medical staff with personal computers, mobile phones, or PDAs. The captured data are analyzed in order to be aware of the continuous evolution of the patients and the environment, as well as for early detection of alarming situations (alarm triggering). This is also the case in homecare applications where mechanisms to define and process the sensing and processing of biomedical and environmental signals are needed. However, sometimes a simple alarm warning might not be enough, and flexibility to evolve as patient status and its environment do is also necessary, indeed often without direct external intervention (adaptability). This might imply starting new applications, stopping, or even modifying existing ones. Thus, to achieve the goal of adaptable monitoring of elderly, applications must be context-aware being able to modify their behavior according to changes on their context.

Besides, these applications are commonly executed in distributed and heterogeneous environments, and mechanisms for managing widespread and specific devices with different capabilities (from embedded devices to those with high processing capacities) are necessary. As they supervise the health of patients, their response must be efficient in order to react as quickly as possible to dangerous situations, so a suitable resource management system is needed, not only because it is essential for dealing with the limitations of embedded systems, but also to ensure efficiency. Preventing service disruptions is also mandatory in order to avoid information losses, especially in emergency cases. Consequently, availability must be guaranteed in case of failure in processing nodes or sensor devices. Finally, other critical aspects are privacy, confidentiality and integrity of the data about patients (safety and security).

Therefore, AAL systems for the elderly raise several challenges for developers that have to be taken into account at the requirements analysis and design phases, and that have to be ensured at runtime [23–25]. There are several works that deal with safety, privacy and security issues related to data storage, processing and transmission. Message encryption using Public Key Infrastructure (PKI) and Secure Socket Layer (SSL) [26], authorization and authentication mechanisms [27,28], and the development of security frameworks [29,30] or safety policies [31] are the most usual solutions.

On the other hand, there are also middleware systems that help an application to interact or communicate with other applications or hardware through networks. These kinds of middleware systems are commonly built over a framework layer which solves ubiquity challenges. Examples of such frameworks are Open Services Gateway Initiative (OSGi) [32], Remote Procedure Call (RPC) [33], Object Request Broker (ORB) [34], Reflection [35] or Foundation for Intelligent Physical Agents (FIPA) [36] compliant frameworks.

Self-adaptive systems are commonly defined in the literature as those capable of automatically modifying themselves in response to changes in their operating environment [37]. This requires self-awareness and context-awareness, *i.e.*, the system must be aware of its own state by means of monitoring both, existing resources and its context. Nevertheless, most of them are ad-hoc solutions that assume stateless applications and, as far as authors know, they do not offer means for defining the application evolution to context changes.

This paper focuses on these issues, adaptability and availability, identifying the needs of the target applications and offering appropriate mechanisms to meet both requirements at the different phases of the application life cycle. In particular, mechanisms for defining, based on the medical expertise, how the application must evolve to context changes as well as mechanisms to manage the application at run-time, assuring that the application is available in case of device failure even for stateful applications. This latter is achieved by means of a multi-agent based middleware (MAS).

Previous works of authors are related to applying modeling techniques for developing service-based applications without the necessity of a central orchestrator [38,39]. Additionally, the preliminary idea of the multi-agent based middleware proposed in this work was presented in [40]. With respect to these previous works, this paper contributes a domain modeling approach that allows systems definition from different points of view. The user view (medical staff) defines, using concepts from the area of expertise, the monitoring of patients and their environment, including the adaptation

of the applications to context changes (patient or environment). The software view guides the software developer in the design and implementation of the components required for providing the medical care specified in the user view. The paper also extends the preliminary middleware to manage events signaling special situations and the associated actions to be taken. Finally, application availability is assured by taking advantage of the mobile nature of agents. This is a generic approach as the middleware offers generic agent templates to be used to define any application that evolves with its context.

The remainder of this paper is as follows: Section 2 presents some related work on both adaptability and availability in home care AAL systems. Section 3 identifies the challenging requirements demanded by homecare applications. This section also includes a brief description of the proposed solution that consists of a domain modeling approach and a MAS middleware. In Section 4 the modeling approach for defining the application dynamic behavior is presented while Section 5 presents the MAS-RECON middleware that provides a set of agent types for implementing flexible homecare applications as well as mechanisms to manage their execution. Section 6 is dedicated to the assessment of the proposal based on the implementation of a healthcare demonstrator and some experimental tests. Finally, Section 7 outlines the most important conclusions and future work.

2. Related Work

This section comprises some research work dealing with the focus of the paper, *i.e.*, adaptability and availability in homecare applications for elderly.

As far as authors know, the majority of works in the literature lack *adaptability* mechanisms, as they focus on alarm triggering in case of danger, asking for medical assistance or warning the patient. In this context, some works provide closed solutions that can be configured by the final user such as [41]. Other works provide means for application design aiming at alarm identification [28,42–46] or at the specification of the responses [47,48], which commonly correspond to warning or alarm triggering. For instance, the Millennium Home System [47] allows defining how to select the best mode of interaction with the user, and whether the resident or an external service have to be warned. In this context, one of the easiest ways of covering a broad range of situations and responses is the use of the event-condition-action (ECA) paradigm [49]. How ECA rules allow defining the actions that have to be executed when certain events are detected is presented in [48].

With respect to alarm identification there exist different approaches in the literature. For example, the CommonSens system [42] proposes an event language to describe events, and the Necessity project [43] presents a rule-based classifier that determines if a situation is normal or abnormal. Some authors make use of modeling methodologies [50] as they allow representing a system at different abstraction levels, hiding irrelevant technical details [28,44,45]. In this sense, the specification and verification approach in [44] combines UML diagrams and formal methods for establishing time requirements associated to events. These design approaches have something in common; they focus on software developers. On the contrary, but also based on modeling techniques, there are works that incorporate domain experts in the system design and development as in [45] where physicians define the conditions to trigger the alerts to display in a view, or in [28] where they model the care process and nurses manually initiate the different actions related to an alarm. The CAALYX system [46] proposes a special purpose language for caretakers to define the clinical rules. These rules detect health alteration by means of observation templates that are customized for patients in the so-called observation patterns. Among the analyzed works, the CAALYX system might be the most similar to the work presented in this paper. However, as far as authors know, it is neither possible to automatically start the execution of new observation patterns as a result of an alert (dynamic adaptation), or to relate changes on the environment with the monitoring of patients.

Related to implementation issues, there is a substantial body of literature on self-adaptive systems based on reconfigurable middleware systems. As previously stated, this kind of middleware systems

are commonly built over a framework layer which solves ubiquity challenges simplifying component management, update and communication.

The THOMAS middleware [27] combines multi-agent technology and service orientation, offering registration mechanisms for services, their implementations and organizations. It allows the organizational structure to be dynamically modifying by creating new ones, or by adding and removing members. However, this capability is restricted to some concrete roles. In the CARISMA project [35], self-adaptation is tackled by defining profiles as fixed sets of actions the middleware should take when a specific event happens. Another approach is based on the so-called sentient objects [51] which are able to take decisions and perform actions. Actions to be performed as a response to context changes can be statically defined as part of the middleware at design time [52], or they can be built at runtime [53,54]. Nevertheless, these approaches are not fully generic as they are presented as part of an application domain and therefore they represent an ad hoc solution to a concrete problem. On the contrary, the iLAND project [55] proposes a general-purpose middleware for real-time systems with time-bounded reconfiguration capabilities. However, it only supports sequential stateless applications and it does not provide support for managing context events.

Application *availability* even in case of device failure is usually managed at the application level, and therefore the application state is implicitly managed by itself. This is the case of the architectures defined in [27,56] offering redundant service providers. The service oriented component model described in [57] provides location independent peer to peer (P2P) communications between components. The GAL platform [58] also defines services as reusable blocks and availability is assured by means of redundancy on services. In the iLAND middleware [55], availability is supported by creating several implementations related to a service. Therefore, the formers present application aware recovery and the latter only supports stateless services recuperation in case of a node failure.

3. Flexibility Requirements for Home Care AAL Systems

For a better understanding of the requirements identified in this section, the next paragraphs describe some use cases related to an old people's home. These use cases have been documented in and inspired by some literature works related to heart rate, blood pressure, oxygen saturation and body temperature monitoring [59–63]. They illustrate simple examples of real use cases being simple enough to represent the flexibility demands of this type of applications.

Use Case 1 (UC1)—Body temperature monitoring. After a surgical operation, the body temperature of a patient is measured four times a day. These values are stored for further analysis. Additionally, if the temperature is over a concrete threshold (according to the patient particularities), the medical staff has to be warned in order to supervise a possible infection.

Use Case 2 (UC2)—Heart rate monitoring. In order to detect a possible heart attack, the pulse rate of a patient is monitored every 10 min. However, if the heart rate trend indicates an abnormal increase (according to the patient particularities), apart from warning the medical staff, the acquisition frequency must be increased for a more detailed monitoring.

Use Case 3 (UC3)—Fire detection. In a nursing home, monitoring the physical environment is crucial. In case of fire, collecting information about health of patients might help emergency services to make decisions on arrival. Thus, buildings are usually equipped with fire detectors and upon the detection of a fire new health monitoring tasks must be launched for every patient, such as pulse rate and oxygen saturation level monitoring.

Use Case 4 (UC4)—Blood pressure monitoring. The main objective of this use case is to monitor the blood pressure of a patient, four times a day. However, as it is presented in [62], blood pressure measures are only relevant if the patient is relaxed. This situation can be checked by means of its pulse rate. Therefore, before taking a blood pressure reading it is necessary to supervise the pulse rate of the patient (one measure every 30 s) until it is relaxed or a maximum waiting time is exceeded. Of course, if the pulse rate or the blood pressure is out of range, medical staff has to be warned.

As it can be concluded from these examples, target applications have three main objectives: (1) monitoring; (2) early recognition; and (3) rapid and suitable reaction. To match these goals, context information is captured by means of sensors that monitor vital functions (all use cases) and acquire environmental measures (UC4), taking into account that each measurement must be performed at the right frequency (temperature is taken every six hours in UC1 whereas heart rate is measure every 10 min in UC2). The processing of these data enables a continuous monitoring of the patient health and their environment, being possible to foresee risky situations and to provide the most suitable assistance in case of emergency. Furthermore, actions for acquiring biomedical sensor measurements and related processing must be customized for every particular elder, although there are similarities among many of them. Indeed, there exist medical guidelines that give support to medical professionals in making general decisions on the treatment of a patient, which, in the end, varies from patient to patient. For instance, pulse measurement can be always carried out in the same manner, whilst a concrete pulse value has a different meaning according to various factors such as the patient age, physical activity, ambient temperature, *etc.*

Target applications supervise dynamic systems that can evolve to dangerous situations. For instance, the pulse rate of a patient increases in case of heart attack. In these situations, besides the usual monitoring and alarm detection, a reaction to the alarm must be defined. Warning medical staff, as in UC1, is the easiest response which is already performed by the works described in the related work section. However, sometimes the application has to evolve in response to relevant changes on its context. As a result, it could be necessary to change the acquisition rate as in UC2 (in case of a possible heart attack), or to initiate the processing of new biomedical values as it is stated in UC3 (new monitoring tasks have to be launched after fire detection) and UC4 (blood pressure monitoring is started). Sometimes, as in UC4 it is necessary to stop current actions (heart rate monitoring is stopped after patient is relaxed).

Finally, continuous monitoring implies to assure application availability even in case of node failure. Furthermore, service recovery has to be application unaware, that is, the application design has not to be altered to match this requirement. Special attention has to be paid to the particular case of those services whose result depends on previous executions (the so-called stateful services). For example, in the UC4 (blood pressure monitoring), several subsequent pulse rates must be analyzed in order to assure the relaxed condition. When a node executing this analysis fails, the recovery process implies restoring the previous pulse rate values. In summary, the main requirements demanded by the target applications are collected in Table 1.

Table 1. Requirements demanded by the target applications and their relation with the proposed use cases.

Requirement Identifier	Requirement Description	Use Cases that Represent It
R1 Personalized sensing and processing	Support for different sensors, customized processing and thresholds.	All
R2 Distributed and heterogeneous environments	Integration of distributed sensors and heterogeneous platforms (resources).	All
R3 Activation and execution types	Actions triggered by time or by event.	All
R4 Adaptability	Context changes awareness: modifying timing properties, launching/stopping applications . . .	UC2, UC3 and UC4
R5 Availability	If a device fails, application must remain unaffected.	All

In order to meet the requirements identified above, this paper proposes to divide a monitoring action into a set of measuring and processing tasks that are customized according to the particular health problems of the patient. These tasks can be executed in distributed and heterogeneous devices and have to be interconnected to achieve the monitoring goal. With this purpose, this paper proposes a

system architecture that consists of a domain modeling approach and a multi-agent based middleware, the so-called MAS-RECON middleware.

The domain modeling approach guides the specification of applications and the implementation of the corresponding components. It has two stakeholders, medical professionals and software developers, and therefore it has been divided in two domains: the user view and the software view. The definition of the user view is the responsibility of the medical professionals as they define the customized treatment for every patient. It consists of a set of interconnected tasks in order to provide the required medical service, which includes the monitoring, alarming situation detection and reaction. On the other hand, software developers are the responsible for the software view which is based on the previous one. It comprises the set of components in charge of acquiring and processing the biomedical and environmental signals. These components are connected following the logic established by the medical staff. Therefore, software developers implement the required medical services and the application logic to connect them. In a sense, the modeling approach allows the medical professionals to specify the software developers what to do and how to do it, by using concepts close to its area of expertise.

The MAS-RECON middleware extends the Java Agent DEvelopment (JADE) framework [64] and manages the execution of applications. In order to implement the application code, software developers are provided with code templates that have to be filled in with the functional specification of the user view. At runtime, the proposed middleware architecture together with the logic added to the code templates and the negotiation capabilities of agents are the means to support the flexibility of applications and fault tolerance. The next sections detail the proposed domain modeling approach and middleware architecture.

4. Domain Modeling Approach for Application Specification

In order to reach a correct and full-customized health monitoring, it is necessary to incorporate domain experts in the system definition and development. With this purpose, this section describes a domain modeling approach that allows defining the whole application abstracting the implementation issues. As previously mentioned, two different but related domains have been identified: the user view and the software view. More precisely, the software view generalizes the user view by extending existing concepts with new properties and by adding new concepts.

4.1. User View

Medical professionals and maintenance staff provide the information related to the *user view* that is constituted by a set of concepts and relationships among them. These concepts allow defining the functional requirements (R1, R2 (Table 1)), the timing requirements (R3) and the dynamism (R4) needed for the health monitoring of patients and the supervision of the environment, from the medical professionals perspective. Availability requirement (R5) is application unaware and thus, it is not covered by the modeling.

4.1.1. Functional Requirements (R1, R2)

This view defines the health monitoring of every patient and the supervision of the environment by means of the *Scenario* concept. Figure 1 illustrates the concept of Scenario through the specification of a nursing home (*System* concept) with three patients (Scenarios).

Health monitoring has to be customized to each patient. Therefore, physicians have to identify which biomedical variables to monitor and how to process them, using the *Application* concept. For example, as it is depicted in Figure 1, the special monitoring for emergency situations described in UC3 is defined for every patient. “Patient 1” represents a resident without any relevant health problem. Its temperature is monitored as it has been operated on. “Patient 2” is related to a resident with hypertension. Thus, its blood pressure has to be controlled as it is explained in UC4. Finally, “Patient 3” refers to a resident with heart disease (UC2).

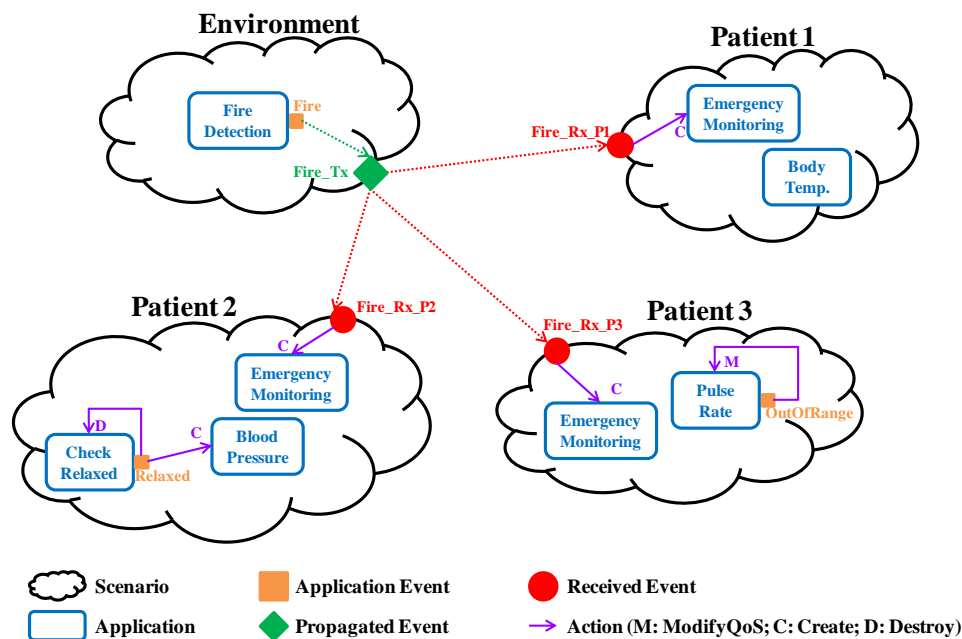


Figure 1. Graphical representation of the user view specification related to a nursing home with three patients.

The monitoring of a biomedical or environmental variables involves several tasks (*AppComponent* concept), including: data acquisition at concrete frequencies from sensors (or extracted from a repository in bulk); and processing activities to obtain useful data for the medical staff. Medical professionals have to describe these tasks (name and *providedServiceDesc* properties) from which software developers implement the needed software components (again, *AppComponent* concept). For example, in order to check the body temperature, four tasks are necessary: one for temperature acquisition (once a day), another for storing these measures in a repository, other one to check if the captured values are out of the normal range of the resident, and the last one to warn the medical staff in case of detecting an abnormal situation.

4.1.2. Timing Requirements (R3)

Non-Functional requirements are collected as properties related to the previous concepts. Indeed, specifying the timing requirements of every monitoring is essential (*timingProps* of the *Application* concept). More precisely, it is necessary to identify when the monitoring has to be activated (activation properties) and how it has to be performed after activation (execution properties). For example, in the case of “Patient 2” blood pressure has to be measured four times a day which implies that it has to be periodically activated every 6 h. However, after activation its pulse rate has to be periodically monitored every 30 s until relaxing (periodic execution), whereas blood pressure is measured just once (one-shot execution). Additionally, there are also configuration parameters (*configParam* property) such as the patient identifier that allow the customization of the health and environment monitoring.

4.1.3. Adaptability (R4)

Lastly, the *Event* concept allows medical professionals and maintenance staff to identify relevant context changes that demand a reaction. Therefore, they have to detail how to detect every relevant situation and how to react to them. Note that, as it is collected in [65], the context term may have very different meanings. In this work, it refers to the health status of a patient and/or the state of the physical environment. Detecting a context change is the result of data processing. For example, in the heart rate monitoring described in UC2, an abnormal increase of the heartbeat is considered a relevant context change (*OutOfRange* event in Figure 1). In the same manner, it is essential for UC3 specifying

under which circumstances the captured environmental signals (smoke, temperature, etc.) are related to a fire (Fire event in Figure 1). In UC4, the instant at which the patient relaxes is relevant (Relaxed event in Figure 1). This is detected after processing several pulse rates, between its maximum heart rate (HR_{max}) and its resting heart rate (HR_{rest}).

On the other hand, specifying how to react against a relevant context change comprises the actions to be performed after its detection (*Action* concept). There are several types of actions and every one refers to an Application, the target of the action from now on. For instance, in UC2, after detecting a risky situation, it is necessary to increase the acquisition frequency (Modify action). Therefore, the target of the action is the monitoring itself. In Figure 1 this fact is represented by a purpled line that starts on the OutOfRange event and which points to the monitoring itself. However, as it is also depicted in Figure 1, once a patient is relaxed (i.e., the Relaxed event is triggered) the actions are to finish pulse rate monitoring (Destroy action) and to start blood pressure monitoring (Create action). Note that in these examples, after detecting a context change of a patient, the actions performed are related to monitoring tasks of the same patient. But sometimes the actions drawn from a context change goes beyond the patient itself, that is, context changes are propagated (*TxScnEvent* concept). This is the case of the fire detection that requires the starting of a particular emergency monitoring for all the patients at the nursing home, as it is illustrated in Figure 1 (Fire_Tx). Therefore, a scenario can propagate events and it can also receive events propagated by other scenarios (*RxScnEvent* concept). The latter also has associated actions whose target application belongs to the scenario itself.

4.2. Software View

The software view inherits the user view and extends it to define the tasks specified by the medical professionals. In this context, an Application is defined as a set of components (AppComponent concept) that cooperate to achieve application tasks (R1, R2). Therefore, at the software view an application component represents a set of monitoring activities (service unit, from now on), together with the application logic (which data has to be sent, when and to which components) and the event-triggering logic (detection of relevant context changes and reaction), previously defined at the user view. A service unit requires a set of input parameters to offer its service and it provides a set of output parameters after its execution. From now on, component parameters and service unit parameters are interchangeably used (*Parameter* concept). Figure 2 illustrates the CheckHRTrend component belonging to UC2. It detects if the heart rate evolves to exceed the normal range of the “Patient 3” (isOut parameter). It requires a pulse value (Pulse parameter) and the time instant (TimeStamp parameter).

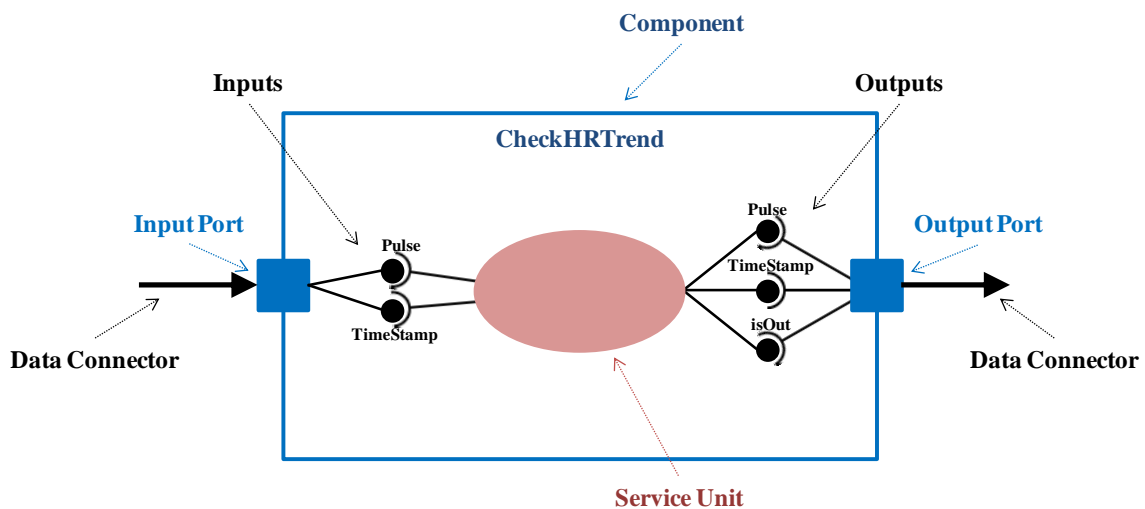


Figure 2. Detailed representation of the CheckHRTrend component.

Application components are also characterized by the timing properties and configuration parameters (R3). For example, a periodic pulse rate monitoring means that the component in charge of the sensor reading has to be periodically executed, but other components will be executed on demand, *i.e.*, after data reception. Additionally, it is also necessary to indicate if the service unit requires additional initialization or finalization actions, and if its execution depends on the result of previous executions (stateful component).

Application components cooperate by exchanging all the data necessary to provide their service, *i.e.*, by connecting the input parameters of a component with the output parameters of its predecessors. With this purpose, components are provided with an input port (*InputPort* concept) and/or an output port (*OutputPort* concept), linked through connectors that collect the exchanged data (*DataConnector* concept). Thus, ports encapsulate the interactions with the service unit and with other components.

More precisely, the input port is in charge of receiving data from predecessors, providing the service unit with the necessary input parameters. Similarly, the output port collects the output parameters resulting from the service unit execution, delivering them to the follower components. Every input parameter received by an input port through a data connector has a peer connection with the corresponding output parameter sent by an output port through it. It is important to remark that software developers have to check that both data-types are compatible in order to set these connections (*DataConnection* concept). The CheckHRTrend component depicted in Figure 2 has an input port to receive its inputs through the incoming data connector whose source is the Acquisition component, as it is illustrated in Figure 3. Similarly, it has an output port to send part of the provided output parameters to a subsequent component, through an outgoing data connector whose target is the Warning component. Data connections are established between the output parameters of the Acquisition component and the input parameters of the CheckHRTrend component, as well as between the input parameters of the Warning component and the output parameters of the CheckHRTrend component.

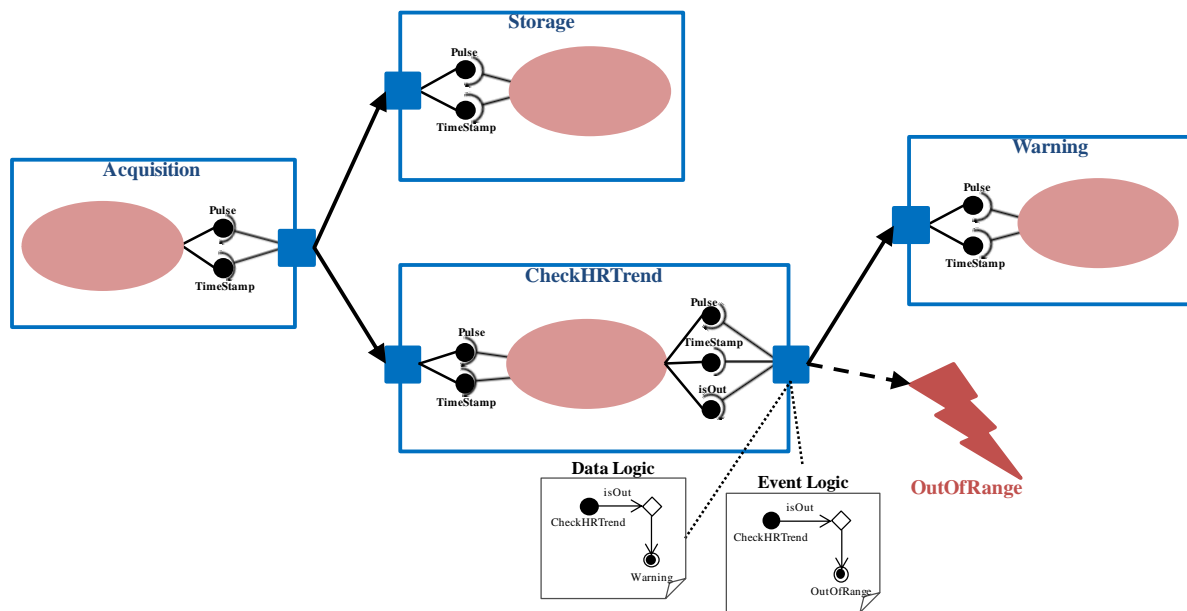


Figure 3. Definition of the application for heart rate monitoring.

Cooperation among components can be led by certain logic, which implies that interface compatibility must be considered from a global point of view. In particular, two different output logic types have been identified (logic property related to the OutputPort concept): Default and Customized. The Default logic implies that the outputs of the service unit are always sent to all followers. In the

example of Figure 3, the acquired pulse values are always sent to be stored (Storage component) and to be analyzed (CheckHRTrend component).

In order to take into account other possible cases, for example when data are delivered under a condition, the customized logic has been defined. This logic is represented by the *DataLogic* concept that is expressed by means of a UML activity diagram associated to the output port. The 'Initial Node' corresponds to the current component. "Control flows" are based on expressions containing output parameters of the component. And every "Activity Final Node" refers to a subsequent component. In the example presented in Figure 3, when heart rate tendency is abnormal, medical staff is warned through the Warning component. This logic is depicted in the "Data Logic" activity diagram attached to the output port of the CheckHRTrend component.

All the concepts introduced up to now allow expressing the R1, R2 and R3 requirements. On the one hand, they enable the definition of health monitoring customized to patients with the associated timing properties. On the other hand, combining measurements and processing is met as applications are decomposed in components that can be executed in different nodes.

In order to consider the adaptability needs derived from relevant context changes (R4 requirement), the proposed software view extends the user view founded on the idea of the ECA rules. It provides mechanisms for defining how to detect a relevant context change and how to react to it, following the specifications of medical professionals. The detection of context changes is part of the processing (service unit) a component performs, giving the result in any of its output parameters. Thus, the component has an event port (*EventPort* concept) with an activity diagram associated, represented by the *EventLogic* concept. This activity diagram is similar to the one for data logic, but in this case the "Activity Final Node" represents the event to trigger when a context change is detected. In Figure 3, there is an "Event Logic" diagram associated to the event port of the CheckHRTrend component. It represents that a risky trend of the heart rate triggers the OutOfRange event.

The Action concept that represents the actions triggered by events has been also extended. In the Create action, some of the new application components can be started with an initial execution state which is obtained from the execution state of a component of the current application (*stateInfo* property). In the Modify action the new timing properties of the application have to be indicated. Note that some of these actions must be executed following a concrete order (sequence property) whereas others can be executed as decided by the middleware.

4.3. Meta-Model

All the concepts described in the previous sections, as well as the relationships and restrictions among them are presented in Figure 4. Concepts are depicted by means of rectangles. Relationships are classified into four groups: (1) composition (black diamond). For example, a scenario for a patient is composed by a set of health monitoring applications; (2) extension (white arrow). For instance, the Create action extends the abstract Action by adding new properties; and (3) dependency, to state that a concept must be aware of another. For example, the logic for event triggering (*EventLogic*) is based on the output parameters of the application component; and (4) association, to reference other concepts. For instance, actions are associated to applications. Finally, restrictions are represented by means of the multiplicity associated to the relationships among concepts.

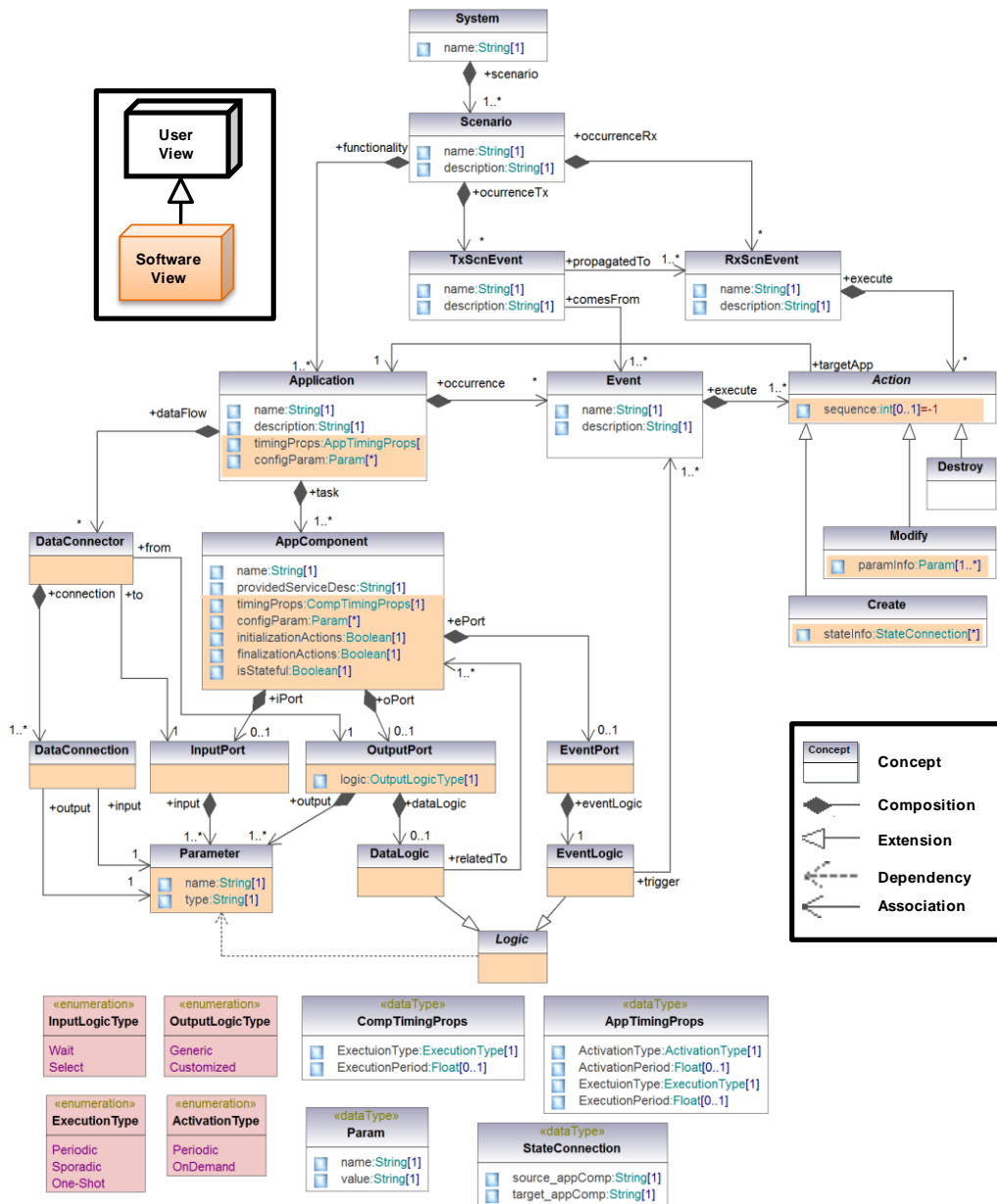


Figure 4. Meta-model of the domain modeling approach for application specification.

5. MAS-RECON Middleware

This section presents the MAS-RECON middleware, a multi-agent based middleware in charge of managing the execution of homecare applications for the elderly modeled in the previous section. The domain modeling approach allows medical professionals to specify the functionality of these applications and their adaptability needs to evolve to context changes. Therefore, the middleware must provide mechanisms for implementing the functionality (meeting R1, R2, R3 requirements), by managing the execution (synchronous and on demand) and communication of application components. It must also provide flexibility mechanisms to enable the adaptation at runtime (R4 requirement). This is met through the event concept implementation, and to assure application unaware availability in case of node failure (R5 requirement). In particular, the proposed availability mechanism is based on a negotiation process among the nodes for finding the most suitable node to hold a new instance of a failed component.

Taking all these demands into account, Figure 5 depicts the proposed middleware architecture founded on the JADE framework. JADE is a software framework that facilitates the development of interoperable intelligent multi-agent systems. The JADE framework has been extended with the new modules depicted at the upper part of Figure 5 in order to meet the requirements identified in Section 3. (1) a Middleware Manager (MM) which is the main system orchestrator; (2) an Application Manager (AM) module per application, in charge of managing the life-cycle of its components as well as their execution state; (3) a Node Agent (NA) module per node that provides runtime information about the node that is useful for availability support; (4) an Event Manager (EM) module per event that manages all its related actions. Each middleware module is implemented by an agent running in the multi-agent system.

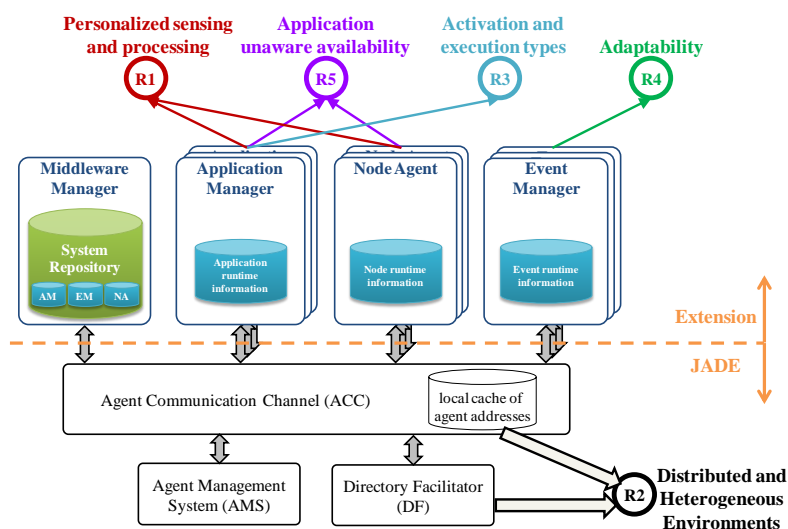


Figure 5. Architecture of the multi-agent based middleware.

5.1. Functional and Timing Requirements (R1, R2 and R3)

The JADE framework is a FIPA compliant agent framework fully developed in the Java programming language. The FIPA foundation promotes agent-based technology and the interoperability of the FIPA standard with other technologies. A FIPA compliant infrastructure must support agent management by means of the following modules (see bottom part of Figure 5): the Directory Facilitator (DF), the Agent Management System (AMS), and the Agent Communication Channel (ACC). According to the FIPA specification, there must be at least one DF agent in the platform, which supplies the yellow pages where agents can register offered services or look for required services. The AMS manages the agent creation, removal and migration. The ACC supports interoperability within and across different platforms. Finally, the so-called Internal Platform Message Transport (IPMT) provides a message routing service for agents on a particular platform.

The domain modeling approach allows distribution as applications are defined as sets of interconnected components that comprise the provided service, the logic to connect them and the logic for event triggering. In this context, the underlying JADE framework allows fulfilling the R2 requirement (distributed and heterogeneous environments) as every component instance is an agent running on the system and agents are mobile in nature. Additionally, as Java is platform independent and JADE can run even in embedded devices, the proposed middleware supports different types of nodes with different capabilities, from embedded devices such as mobile phones and sensors to those with high processing capacities.

Additionally, as these distributed agents cooperate by exchanging messages, three FIPA compliant ontologies have been defined in order to support communications among agents: (1) Data ontology for message exchange containing the data necessary to provide a medical service or environment

supervision, such as sensor values and processing results; (2) Command ontology for control commands that allow agents or technicians interact with the middleware modules and vice versa; (3) State ontology for updating the execution state of an agent (value of relevant variables).

The MM module manages information about the whole system which is collected in the so-called System Repository. The hierarchical structure of this repository is presented in Figure 6. It contains runtime information about the running applications, the triggered events and booted nodes. This part of the repository is distributed throughout the corresponding middleware modules. It also contains design information including the physical nodes and the data coming from the software view.

Physical nodes are the hardware devices where the instances of application components run. This includes access to sensors, actuators and processing units. Every node contains an instance of the NA module which provides physical information (core number, storage and memory capacity, network speed, CPU score and platform) and runtime information (CPU usage, free memory) about the node. A NA registers itself automatically at boot time, providing the MM with its resource capabilities (highlighted in yellow at Figure 6). They also perform the negotiation process when it is required by the AM.

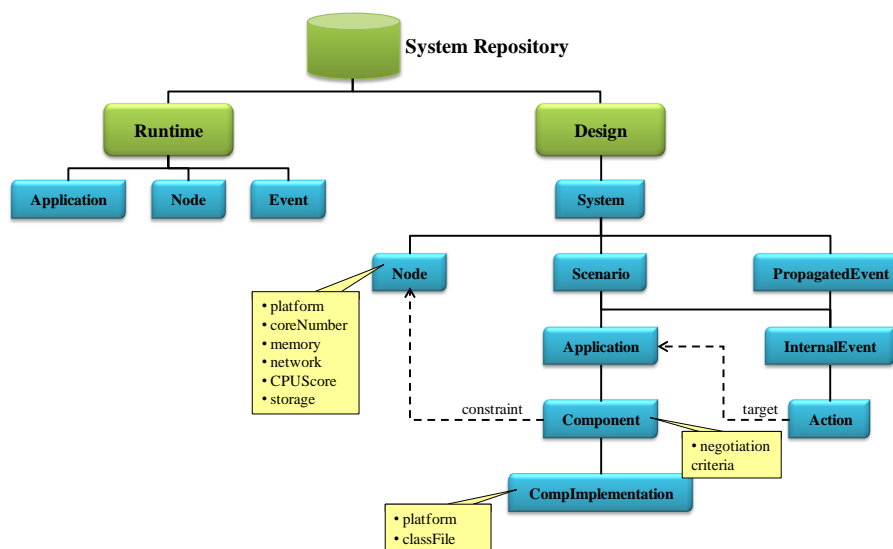


Figure 6. Structure of the System Repository at the Middleware Manager (MM) module.

Software developers are the responsible for registering the information related to the application itself: the system, the scenarios, the applications that belong to each scenario and their components. Events and the actions to be performed have to be also registered. Two types of events have been considered: internal events (InternalEvent) and propagated events (PropagatedEvent). Internal events belong to a scenario, and their actions refer to applications of the same scenario: Event concept in Figure 4 and the events received by the scenario and that have been propagated by other ones (RxScnEvent concept in Figure 4). In the middleware, the events propagated among scenarios (TxScnEvent concept in Figure 4) are composed by the set of internal events associated. For example, in the nursing home system depicted by Figure 1, the FireTx event is a propagated event, whereas OutOfRange, Relaxed, FireRx_P1, FireRx_P2 and FireRx_P3 are internal events.

Additionally, the structure of the System Repository takes into account that a component can be implemented in several ways (CompImplementation). Note that it is also possible to restrict the nodes where the instance of a component implementation, component instance or agent from now on, can be executed (constraint). For instance, the software developer may use different platforms or libraries, restricting the available nodes to execute them. The need of a concrete sensor only accessible from a node is another example of constraint. This way a task of the user view is linked to a specific

node, through a component of the software view. It is important to remark that when adding a new node to the system new component constraints should be registered, if needed. Moreover, the skeleton code of these component instances has been fixed in order to match the R1 (personalized sensing and processing) and R4 (application unaware availability) requirements. More precisely, every component instance must implement the FSM represented in the left part of Figure 7, having the following states:

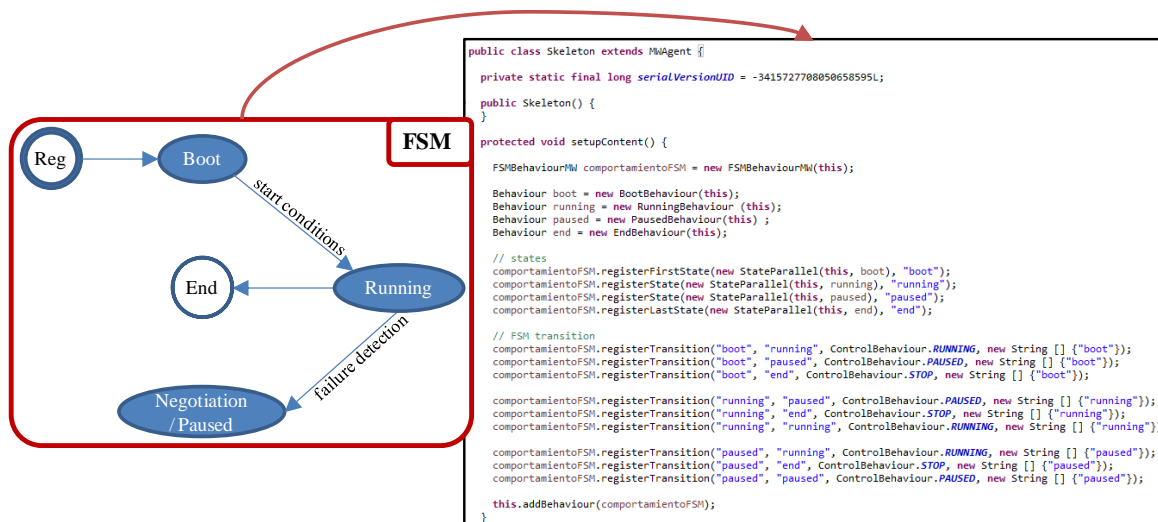


Figure 7. Finite State Machine (FSM) and its Java implementation.

- *Boot*: during this FSM state, the agent waits until its start conditions are met. This allows executing the required initialization actions and synchronized start of agents. When the start conditions are met, the agent switches to the Running FSM state.
- *Running*: in this FSM state, the agent is offering its functionality related to a medical service. Besides, every cycle the execution state is stored at the corresponding AM, for availability purposes.
- *Negotiation/Paused*: when a component failure is detected, the AM forces the agent to this FSM state.
- *End*: during this FSM state the agent finishes its execution which includes the required finalization actions.

The skeleton code derived from this FSM and implemented in Java is also depicted in the right part of Figure 7. Additionally, application components are provided with a control interface through which they receive control commands. The software developer has to customize this skeleton code for every application component founded on the software view of the modeling approach. In particular, if the component requires initialization actions a new Java class that extends the Boot FSM state by including all the needed actions has to be implemented. For example, the pulse oximeter sensor used in the demonstrator must be initialized. Similarly, if the component requires finalization actions a new Java class that extends the End FSM state with these actions has to be implemented. The Running FSM state has to be always customized in order to include the medical service offered, the data logic and the event triggering logic. Therefore, another Java class has to be developed. With this purpose, two templates have been defined according to the activation mode of the agents (R3 requirement):

- (1) Periodic: this template is based on the TickerBehaviour class of JADE. It is used for components that execute the service periodically. Therefore, every cycle they run their functionality, send results, if any, update the execution state and delay until the next activation.
- (2) On demand: this template is based on the CyclicBehaviour class of JADE. It is used for components that execute the service after the reception of a data message. Therefore, they wait for all incoming messages, run the functionality, send results, if any, and update the execution state.

For example, in UC2 depicted in Figure 3, the component in charge of the pulse reading is executed every 10 min (periodic), but the component in charge of warning the medical staff is executed just after receiving input parameters (on demand).

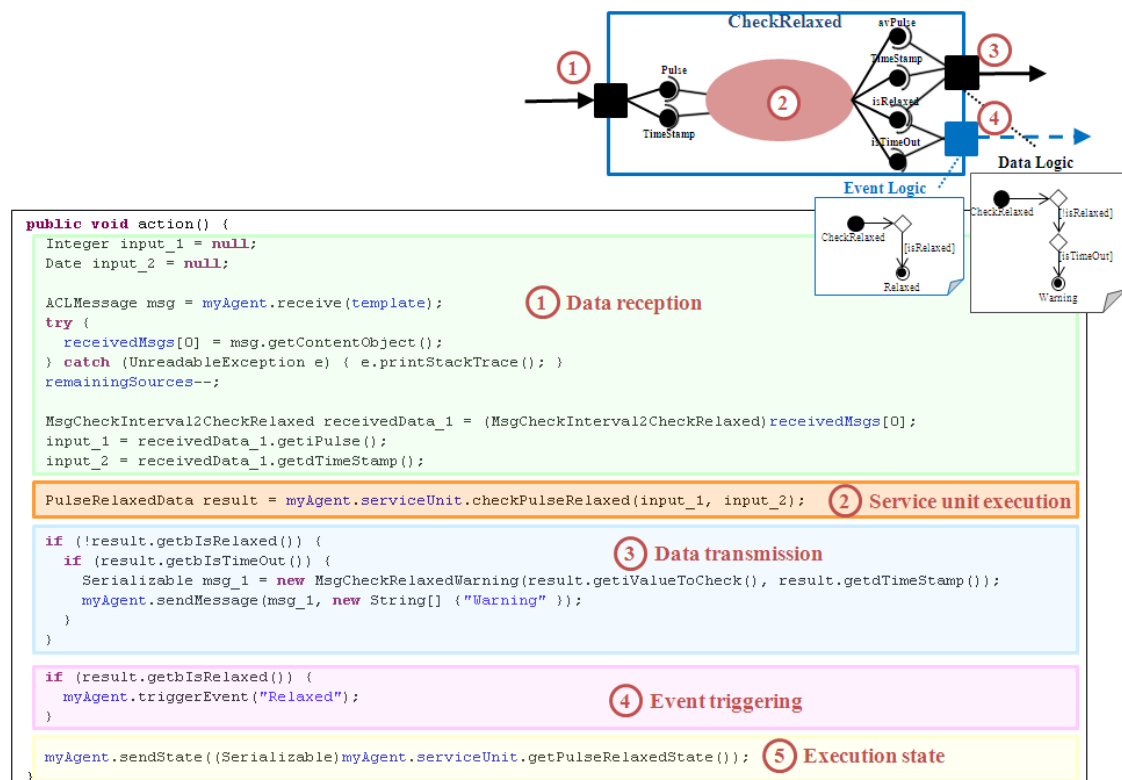


Figure 8. Customization process of the Running FSM state for the CheckRelaxed component.

As an example, Figure 8 presents the customization of the Running FSM state related to the CheckRelaxed component. This component belongs to the Check Relaxed application (UC4, see Figure 1). It receives two input parameters, a pulse value (Pulse) and the measurement instant (TimeStamp). It analyzes the new value together with several previous ones in order to determine if the patient is relaxed or not. Therefore, it provides four output parameters: the average pulse (avPulse), the last measurement instant (TimeStamp), a flag for patient relaxation (isRelaxed), and a flag that indicates if the waiting time has been exceeded (isTimeOut). When the patient relaxes it triggers the Relaxed event (event logic). When the waiting time is over, medical staff is warned (data logic). In summary, the customization process of the Running FSM state comprises the following steps:

- (1) If the component has an Input Port, code for data reception has to be included. Every Data Connector of the modeling approach that ends in the input port is related to a data message received from a previous component. The required input parameters have to be extracted from these messages according to the input logic type defined and the Data Connections established.
- (2) The code for service unit execution is always added. It depends on how the software developer has implemented this functionality.
- (3) If the component has an Output Port, code for data transmission has to be inserted. Every Data Connector of the modeling approach that starts in the output port is related to a data message sent to a subsequent component. The output parameters obtained as a result of the service unit execution have to be grouped according to the Data Connections established, composing all the necessary output messages. Additionally, if the output part has a Data Logic attached, the

associated activity diagram has to be parsed in order to write the necessary conditional statements for data delivery.

- (4) If the component has an Event Port, the code for event triggering has to be added. Similarly, the associated activity diagram has to be parsed to include the conditions that have to be filled to trigger every event. If the event is propagated through scenarios the event included in this code is the corresponding TxScnEvent.
- (5) If it is a stateful component, the code for updating its execution state at the corresponding AM has to be included.

At runtime, the execution of the instances of these developed components is managed by the AM module. The MM deploys as many AM instances as launched applications. Each AM is in charge of supervising the execution of the components associated to an application (R3 requirement). This includes several tasks:

- Components startup, which consists of selecting the appropriate node to hold the component instance, by means of a negotiation process.
- Management of the execution state related to stateful components.
- Management of the component life-cycle. It is aware of the current FSM state of every component instance, and it may force it to pass to a concrete FSM state, if necessary.
- Management of component failure detection, due to a node failure, for example.

5.2. Adaptability (R4)

In order to tackle the adaptability needs (R4 requirement), the events registered in the System Repository are supervised by an EM module. The MM deploys an EM instance for each event. It performs the actions established for the event and it supervises they follow the required order, if necessary. Note that the interaction between a component instance and the EM is through method invocations at the source code. For instance, Figure 9 presents how the EM related to the Relaxed event (depicted in Figure 1) supervises its associated actions. When the checkRelaxed001 component instance detects that the patient is relaxed, it triggers the Relaxed event. This event triggers two actions: one for launching the blood pressure monitoring and the other one for stopping the pulse rate monitoring, both through the corresponding AM. As an example, the figure shows how the AM of Blood Pressure application (AM_BloodPressure) starts one of its component instances (bAcquisition002), and how the AM of the *Check Relaxed* application stops one of its component instances (checkRelaxed005).

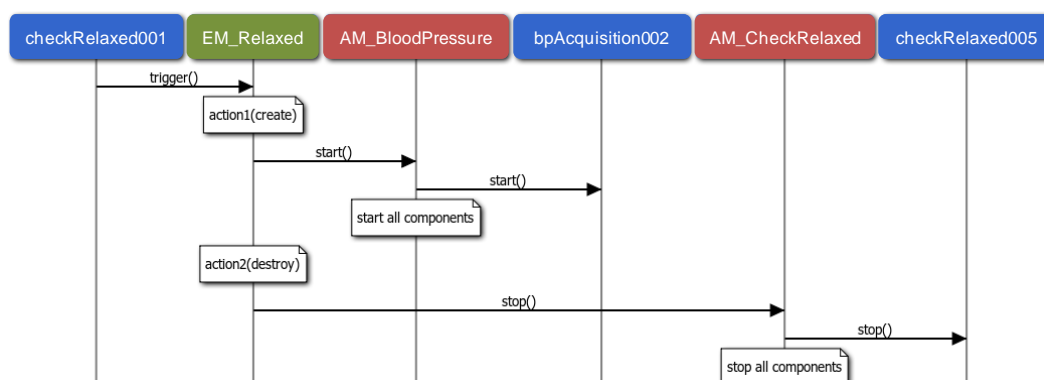


Figure 9. Sequence diagram related to the Relaxed internal event.

Similarly, Figure 10 describes how propagated events are attended. In this case, the fireDetector001 component instance detects a fire and thus, it triggers the Fire event. This event is generated in the Environment scenario and propagated to the other three scenarios. As a result, three internal events

are triggered, Fire_Rx_P1, Fire_Rx_P2 and Fire_Rx_P3 (see Figure 1), each triggering a Create action for launching the emergency monitoring of the corresponding patient. As it is illustrated in the figure, applications are started through the corresponding AM as in Figure 9.

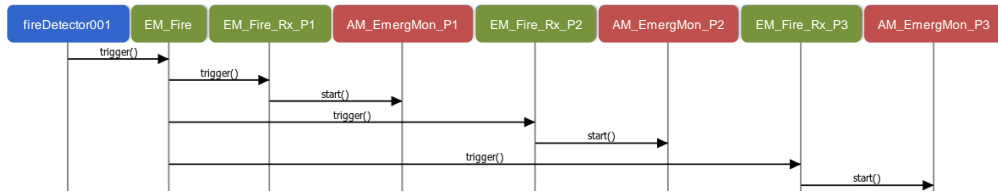


Figure 10. Sequence diagram related to the Fire propagated event.

The benefits of the event manager module are twofold. On the one hand, it optimizes the use of system resources, as upon an event triggering an invocation to the middleware is issued in order to create/destroy/modify applications which in the end implies allocating/de-allocating the corresponding resources. As a result, resources are allocated just when needed. On the other hand, the domain modeling approach provides application independency within scenarios as they are only related through events. Scenarios independence is also supported as they can be connected through propagated and/or received events. At runtime, the EM module implements this independence by executing the actions related to the triggered event. As a result, adding new monitoring applications or adding a new scenario (a new patient) to an already running system does not modify the implementation of the system. Instead, the system extension implies registering the new applications/scenarios and the corresponding events, if necessary, as well as the implementation of the new components.

5.3. Application Unaware Availability for Stateful Applications (R5)

AMs and NAs are the main participants of the middleware support for application unaware availability (R5 requirement). As commented above, the proposed availability mechanism is based on finding the most suitable node to hold a new instance of a failed component. Therefore, on the one hand it is necessary to detect component failures, and on the other hand it is necessary to recover it. As an example, Figure 11 illustrates the recovery of a stateful component. In particular, it is the CheckRelaxed component of the Check Relaxed application (it monitors pulse rate, UC4), whose previous component is the so-called CheckRange.

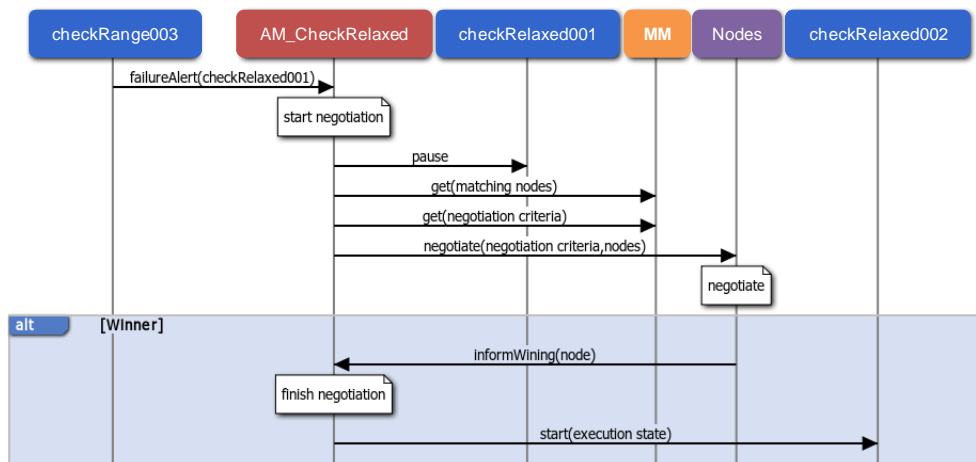


Figure 11. Application unaware availability: failure detection and stateful recovery.

Component failures can be detected in two ways: when the sender of a data message detects that it has not been possible to deliver it, or when a periodic component exceeds the period to refresh its execution state in the AM. In both cases the corresponding AM is notified and the failed component instance is labeled as faulty. This avoids attending to the same failure more than once. After, the AM starts the recovery process. In the example of Figure 11, the checkRange003 component instance detects a component failure as the data message sent to the checkRelaxed001 component instance has not been delivered.

A component recovery starts with a negotiation among all the NAs related to nodes that can hold a new instance. These nodes are selected taking into account the node constraints and the available implementations of the component, and they negotiate according to the negotiation criteria established during the registration. The negotiation criteria can be, for example, the highest free memory or the lowest processor usage. Once there is a winner NA, the AM finishes the negotiation process, and deploys a new component instance on the winning node, initialized with the last execution state. In the example, as a result of the negotiation process the checkRelaxed002 component instance is started with the last execution state updated by the failed checkRelaxed001 component instance.

6. Assessment

This section presents the feasibility of the proposed solution in order to cope with the demands of homecare applications, through its feasibility to deal with the requirements identified. On the one hand, the proposal design is validated by means of a homecare demonstrator. On the other hand, its runtime performance is evaluated by means of a set of experimental tests. More precisely, these tests aim at evaluating the adaptability and availability mechanisms offered by the MAS-RECON middleware. Finally, the main benefits and limitations are highlighted.

6.1. Homecare Demonstrator

A homecare demonstrator that includes the proposed use cases has been implemented, namely, the nursing home represented in Figure 1. Therefore, there are three residents: Patient 1 has no serious health problems; Patient 2 suffers from high blood pressure, so s/he requires blood pressure supervision four times a day; Patient 3 suffers from heart disease, so s/he is provided with continuous heart rate monitoring (every 10 min). On the other hand, the building is equipped with a fire detection system based on the temperature and CO₂ concentration.

From the specification point of view, it is a system composed of four scenarios: three patients and the environment. The Environment scenario consists of an application for fire detection that triggers the Fire event, if detected. This event is propagated to the other three scenarios. Therefore, all the patient scenarios receive a propagated event that launches a concrete application for health monitoring in emergency situations (Emergency Monitoring). In the Patient 2 scenario, the Check Relaxed application triggers the Relaxed event when the patient is relaxed, launching the blood pressure monitoring (Blood Pressure application) and stopping itself. In the Patient 3 scenario, there is an application for pulse rate monitoring.

The prototype demonstrator consists of biomedical and environmental sensors, and processing units. For health monitoring purposes, the biometric shield for Arduino and Raspberry Pi, the so-called e-Health Sensor Platform V2.0., was used [66] (see Figure 12). Every patient is provided with a health sensor shield mounted over a Raspberry Pi. More precisely, it offers a body temperature sensor, a pulse oximeter (SPO₂) for pulse rate, and a sphygmomanometer for blood pressure. The environment supervision is performed through temperature and CO₂ sensors mounted over a waspmote [67]. The processing tasks can be executed in four PCs.



Figure 12. Infrastructure of the healthcare demonstrator: *e*-Health Sensor Platform V2.0., gas sensors kit and processing units.

From the implementation and deployment point of view, all the application components have been developed in Java programming language. There is a repository for recording information about patients such as personal data (identifier, name, surname, age, sex . . .) and medical data according to their health problems. For example, Patient 3 is characterized by her/his maximum heart rate (HR_{max}), its resting heart rate (HR_{rest}), and its normal range of body temperature. Furthermore, it also stores the historic measures of patients. The patient repository has been implemented by means of the native XML eXist database [68]. The MM, the AM instances and the EM instances run in the same PC. Agents related to application components that manage biomedical sensors are restricted to the corresponding Raspberry Pi. Finally, for availability purposes, agents related to the other application components can be deployed in any of the four PCs.

Taking into account that the use cases illustrate all the requirements identified in Section 3, this homecare demonstrator allows:

- Validation of the Domain Modeling Approach presented in Section 4 as every use case has been designed and developed following it. Note that in this homecare demonstrator there are neither real patients nor medical professionals involved.
- Assessing the middleware architecture design and the services it offers: adaptability to context changes (event management), availability (failure detection and negotiation-based recovery), stateful component management, and registration (system repository).

6.2. Runtime Performance

Runtime performance has been assessed regarding the two main goals of the paper: adaptability and availability. In particular, adaptability is evaluated in terms of the reaction time to adapt to a context change (a change on the health status or environment conditions) whereas availability is assessed according to the recovery time under a failure.

Both parameters are tested by using similar experiments. The starting point is a very simple and sequential application that captures a sensor value, processes it and shows the result. In both cases the number of available nodes to hold component instances is incremented. For availability tests the number of processing tasks is also increased, *i.e.*, the number of components of the application. However, for adaptability, the number of actions triggered by the event is increased. In order to avoid that the different processing capacities of nodes interfere the analysis of the results, and taking into

account that in a real scenario there are many devices with limited resources, all the nodes in the experiment are Raspberry Pi.

Regarding availability, Figure 13 shows the recovery time of the different tests. This time ranges from a node failure to the recovery of all the affected component instances. As expected, the recovery time increases with the number of nodes and components. In fact, the recovery time increases almost proportionally to the number of nodes, as more nodes participate in the negotiation and due to the low processing capacities of the Raspberry Pi, this handicaps the negotiation processes. Similarly, recovery time also augments with the number of application components when more instances are affected by the node failure. However, taking into account that the worst case is about 2 s and that the most restrictive application evolves at 30 s (Check Relaxed application), the time delay is acceptable if compared with the benefits achieved. Additionally, this worst case corresponds to applications whose component instances can run in five different nodes, which is unusual as two available nodes are commonly enough.

As far as adaptability is concerned, Figure 13 depicts the reaction time in milliseconds since an event is triggered until all its associated actions have been performed. For simplicity, all the actions triggered by the event are Create actions. Therefore, the resulting time includes the startup of the applications. Again, as expected, the number of nodes and the number of actions increase the reaction time.

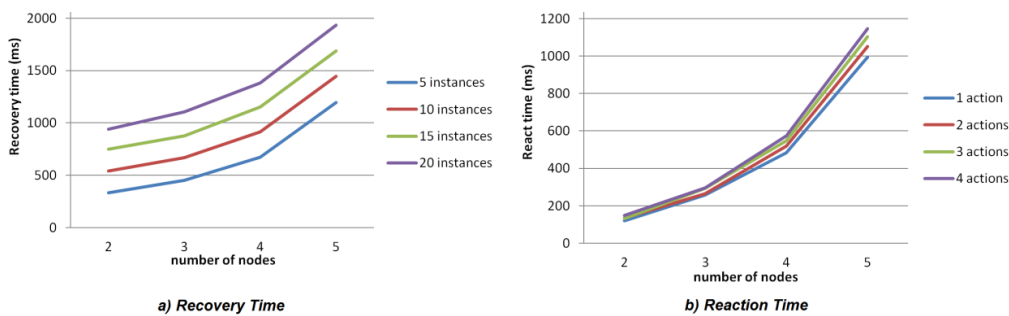


Figure 13. (a) Availability metrics: recovery time; (b) Adaptability metrics: reaction time.

In order to identify availability limitations another test has been performed in a PC. The same application of eight components has been created twice: the first time under normal conditions of CPU load, and the second time after significantly increment the CPU load (up to 80%). As a result, the availability performance has been negatively affected. Figure 14 depicts the number of threads in the node (every running agent is a thread). As it is observed, the start time for the same application increases about a 28% because negotiations among nodes are slower. These metrics have been captured by means of the VisualVM GPL software.

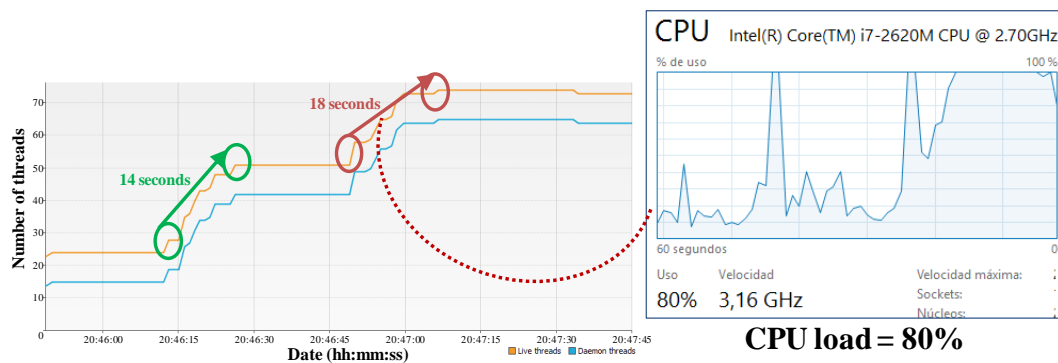


Figure 14. CPU load vs. application start time.

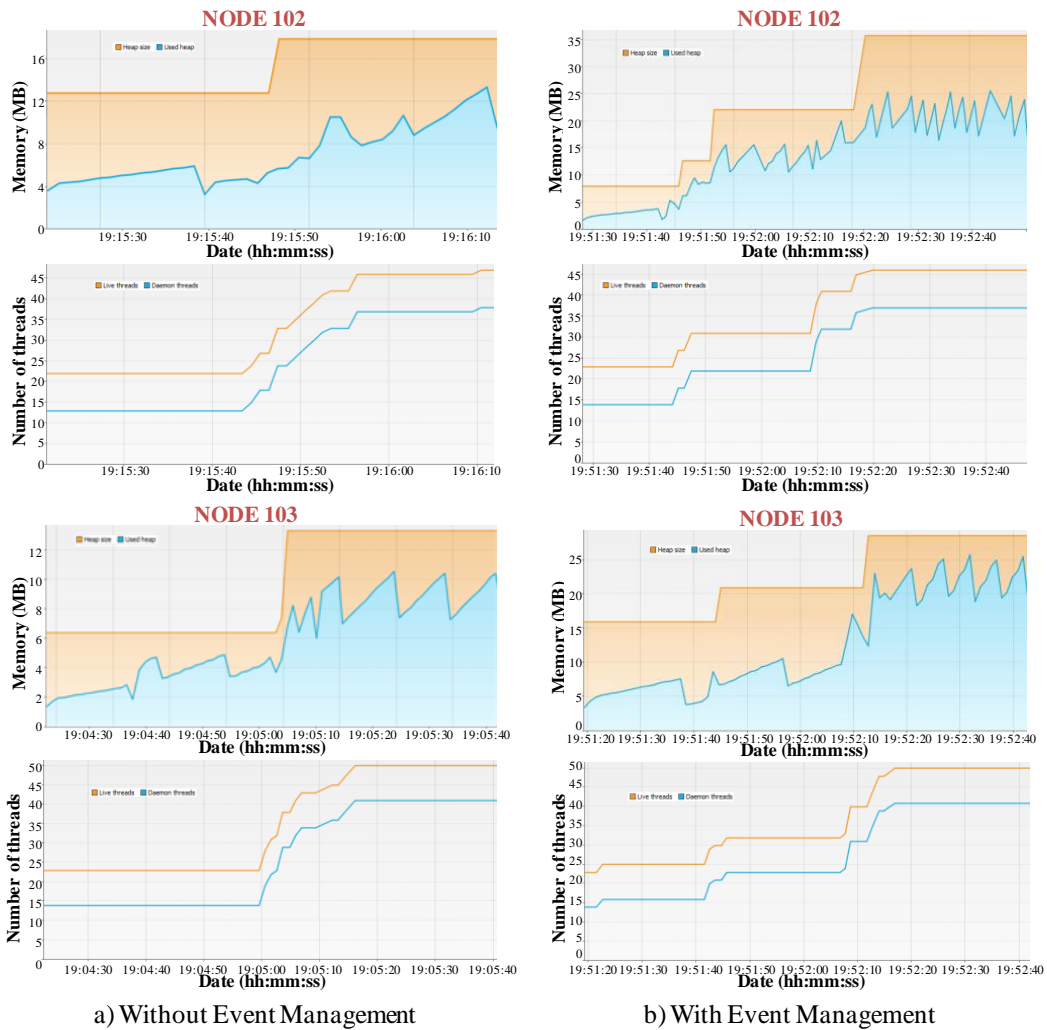


Figure 15. Resource usage in terms of memory usage. (a) without event management; (b) with event management.

As it has been previously stated, resource optimization is one of the main benefits of the event management. In this sense, resource consumption in terms of memory load has been analyzed by means of an application (composed by 18 components) that, after detecting a relevant context change, creates other five applications of 18 components. The component instances of these applications are deployed in four nodes. Figure 15 compares two different tests (related to two of the available nodes). In every graphic, the upper part represents the memory consumption, the orange line refers to the Java Virtual Machine (JVM) heap whereas the blue line is related to the memory used by the loaded objects (here, the JVM garbage collector activations to free memory are noticed). The bottom part depicts the number of threads on the node:

- (a) Without event management: The six applications are started from the beginning. When the first application detects the context change, it sends a data message to the first component of the rest applications in order to activate them. This implies that memory resources are allocated from the start. As a result, in both nodes the amount of memory does not change after the start.
- (b) With event management: the first application triggers an event that is managed by an EM module that performs five Create actions. In this case, there is an initial amount of memory allocated, and after the event triggering, the amount of allocated memory increases.

These metrics prove that events management improves resource usage, which is very useful when resources are limited. However, it implies more reaction time as application components are started after event triggering. More precisely, when there is no event management, reacting to an event by means of an application creation just involves the synchronized start of all the application components, as all have already executed the needed initialization actions.

These tests also show the good performance of negotiation mechanisms. In fact, as the negotiation criterion is the “highest free memory”, all the component instances are similarly distributed among the available nodes. This is showed in the bottom part of the graphics in Figure 15.

7. Conclusions and Future Work

This paper presents a solution for the design, implementation and management of homecare applications for elderly. The proposed system architecture consists of a domain modeling approach and a multi-agent based middleware and it provides mechanisms to tackle their flexibility demands to adapt their behavior according to changes on their context (patient health status or environment conditions) and to avoid service disruption.

The use of domain modeling techniques allows defining applications from different points of views, each gathering the information relevant to it. As a result, the proposed modeling approach allows medical staff to design a personalized monitoring of the health status of patients and environmental conditions. It takes into account adaptability needs from the design phase as it is possible to identify relevant context changes, defining how to detect and how to react to them (user view). Additionally, it guides software developers in the implementation of all the needed software components which contain not only the medical service execution but also the logic for data exchange and the logic for event triggering (software view).

At runtime, multi-agent technology has been adopted to convert components into intelligent entities. In this context, the proposed MAS-RECON middleware has extended the JADE framework in order to manage the execution of these applications, providing mechanisms that allow performing adaptation and that assure availability even for stateful applications. More precisely, the Event Manager module controls all the actions related to an event. As a result, an optimized resource usage is achieved. Availability is assured by recovering the execution of the failed component instance in the most suitable node. This is possible due to the failure detection, stateful recovery and negotiation mechanisms provided by the Application Manager and the Node Agent modules.

The feasibility of the proposal has been proved by means of a healthcare demonstrator based on a nursing home. Several representative use cases have been identified and implemented. Experimental results show that recovery time (availability) and reaction time (adaptability) are affected when the number of nodes that can hold component instances increase or when the number of actions triggered by an event increases. Furthermore, supporting adaptability and availability implies an extra time that is acceptable if compared with the benefits achieved: maintaining application state and resource optimization.

However, the middleware architecture does not support fault tolerance. For example, if an AM fails, the runtime data and execution state related to its application components are lost. The middleware lacks of admission control mechanisms to assure that enough resources are available as the system grows. Therefore, further work is aimed at exploring the distribution of the system repository for improving fault tolerance of the middleware modules, and implementing the admission control. Additionally, proactive mechanisms will be also added in order to match Quality of Service (QoS) parameters. For example, load balancing mechanisms for achieving energy efficiency at node level, or unbalancing mechanisms for energy efficiency at system level (using the least number of nodes). Additionally, as it has been proved in the assessment section, limited resources decrease the middleware performance due to slower negotiation actions. Thus, future work is also focused on supporting flexible QoS for non-critical applications. Finally, Model Driven Engineering techniques will be explored as they allow automating application design and the code generation process.

Acknowledgments: This work was financed in part by the University of the Basque Country (UPV/EHU) under project UFI 11/28, by the Regional Government of the Basque Country under Project IT719-13, and by the MCYT&FEDER under project DPI 2012-37806-C02-01.

Author Contributions: Aintzane Armentia is in charge of the multi-domain modeling approach and the application domain, designing and implementing the different use cases. Unai Gangoit has designed and implemented the MAS-RECON middleware. He has also taken part in the development of use cases and in performing the assessment metrics. Rafael Priego has participated in the design, development and implementation of the demonstrator. Marga Marcos supervises all the research work, whereas Elisabet Estévez is mainly focused on supervising the application domain and modeling approach.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. World Health Organization. Global Health and Aging. Available online: http://www.who.int/ageing/publications/global_health/en/ (accessed on 25 March 2015).
2. World Health Organization. Report of the first WHO Global Forum on Innovations for Ageing Populations. Available online: http://www.who.int/kobe_centre/publications/GFIAP_report.pdf?ua=1 (accessed on 25 March 2015).
3. United Nations. World Population Ageing: 1950–2050. Available online: <http://www.un.org/esa/population/publications/worldageing19502050/> (accessed on 25 March 2015).
4. European Commission. Seventh Framework Programme. Available online: http://ec.europa.eu/research/fp7/index_en.cfm (accessed on 9 April 2015).
5. European Commission. HORIZON 2020—WORK PROGRAMME 2014–2015—Health, Demographic Change and Wellbeing. Available online: <http://ec.europa.eu/programmes/horizon2020/en/h2020-section/health-demographic-change-and-wellbeing> (accessed on 11 April 2015).
6. AMBIENT ASSISTED LIVING JOINT PROGRAMME/ICT for Ageing Well. Available online: <http://www.aal-europe.eu> (accessed on 22 September 2014).
7. U.S. Department of Health & Human Services. Administration on Aging. Available online: <http://www.aoa.gov/> (accessed on 30 April 2015).
8. World Health Organization. Active Ageing: A Policy Framework. Available online: http://whqlibdoc.who.int/hq/2002/WHO_NMH_NPH_02.8.pdf?ua=1 (accessed on 25 March 2015).
9. Woodward, C.A.; Abelson, J.; Tedford, S.; Hutchison, B. What is important to continuity in home care? *Soc. Sci. Med.* **2004**, *58*, 177–192. [[CrossRef](#)]
10. Varshney, U. Pervasive healthcare and wireless health monitoring. *Mob. Netw. Appl.* **2007**, *12*, 2–3, 113–127. [[CrossRef](#)]
11. Memon, M.; Wagner, S.R.; Pedersen, C.F.; Aysha Beevi, F.H.; Hansen, F.O. Ambient Assisted Living healthcare frameworks, platforms, standards, and quality attributes. *Sensors* **2014**, *14*, 4312–4341. [[CrossRef](#)] [[PubMed](#)]
12. Ni, Q.; García Hernando, A.B.; de la Cruz, I.P. The Elderly’s Independent Living in Smart Homes: A Characterization of Activities and Sensing Infrastructure Survey to Facilitate Services Development. *Sensors* **2015**, *15*, 11312–11362. [[CrossRef](#)] [[PubMed](#)]
13. De Silva, L.C.; Morikawa, C.; Petra, I.M. State of the art of smart homes. *Eng. Appl. Artif. Intell.* **2012**, *25*, 1313–1321. [[CrossRef](#)]
14. Chan, M.; Estève, D.; Escriba, C.; Campo, E. A review of smart homes- present state and future challenges. *Comput. Methods Progr. Biomed.* **2008**, *91*, 55–81. [[CrossRef](#)] [[PubMed](#)]
15. Cook, D.J.; Augusto, J.C.; Jakkula, V.R. Ambient intelligence: Technologies, applications, and opportunities. *Pervasive Mob. Comput.* **2009**, *5*, 277–298. [[CrossRef](#)]
16. Nasir, A.; Hussain, S.I.; Soong, B.; Qaraqe, K. Energy Efficient Cooperation in Underlay RFID Cognitive Networks for a Water Smart Home. *Sensors* **2014**, *14*, 18353–18369. [[CrossRef](#)] [[PubMed](#)]
17. Blasco, R.; Marco, Á.; Casas, R.; Cirujano, D.; Picking, R. A Smart Kitchen for Ambient Assisted Living. *Sensors* **2014**, *14*, 1629–1653. [[CrossRef](#)] [[PubMed](#)]
18. Nef, T.; Urwyler, P.; Büchler, M.; Tarnanas, I.; Stucki, R.; Cazzoli, D.; Müri, R.; Mosimann, U. Evaluation of Three State-of-the-Art Classifiers for Recognition of Activities of Daily Living from Smart Home Ambient Data. *Sensors* **2015**, *15*, 11725–11740. [[CrossRef](#)] [[PubMed](#)]

19. Mozer, M.C. The neural network house: An environment that's adapts to its inhabitants. In Proceedings of the AAAI Spring Symposium on Intelligent Environments, Palo Alto, CA, USA, 23–25 March 1998; pp. 110–114.
20. Helal, S.; Mann, W.; El-Zabadani, H.; King, J.; Kaddoura, Y.; Jansen, E. The Gator tech smart house: A programmable pervasive space. *Computer* **2005**, *38*, 50–60. [[CrossRef](#)]
21. Tapia, E.M.; Intille, S.S.; Larson, K. Activity Recognition in the Home Using Simple and Ubiquitous Sensors. In Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications, Orlando, FL, USA, 14–17 March 2004; pp. 158–175.
22. Kidd, C.D.; Orr, R.; Abowd, G.D.; Atkeson, C.G.; Essa, I.A.; MacIntyre, B.; Mynatt, E.; Starner, T.E.; Newstetter, W. The aware home: A living laboratory for ubiquitous computing research. In Proceedings of the Second International Workshop CoBuild, Pittsburgh, PA, USA, 1–2 October 1999; pp. 191–198.
23. Rashidi, P.; Mihailidis, A. A survey on ambient-assisted living tools for older adults. *IEEE J. Biomed. Heal. Inf.* **2013**, *17*, 579–590. [[CrossRef](#)]
24. Nehmer, J.; Becker, M.; Karshmer, A.; Lamm, R. Living Assistance Systems—An Ambient Intelligence Approach. In Proceeding of the 28th International Conference on Software Engineering, Shanghai, China, 20–28 May 2006; pp. 43–50.
25. Becker, M. Software Architecture Trends and Promising Technology for Ambient Assisted Living Systems. In Proceedings of Dagstuhl Seminar, Dagstuhl, Germany, 1–6 June 2008; pp. 1–18.
26. Corchado, J.M.; Bajo, J.; Abraham, A. GerAmi: Improving Healthcare Delivery in Geriatric Residences. *IEEE Intell. Syst.* **2008**, *23*, 19–25. [[CrossRef](#)]
27. Bajo, J.; Fraile, J.A.; Pérez-Lancho, B.; Corchado, J.M. The THOMAS architecture in Home Care scenarios: A case study. *Expert Syst. Appl.* **2010**, *37*, 3986–3999. [[CrossRef](#)]
28. Stav, E.; Walderhaug, S.; Mikalsen, M.; Hanke, S.; Benc, I. Development and evaluation of SOA-based AAL services in real-life environments: A case study and lessons learned. *Int. J. Med. Inform.* **2013**, *82*, e269–e293. [[CrossRef](#)] [[PubMed](#)]
29. Su, C.J.; Wu, C.Y. JADE implemented mobile multi-agent based, distributed information platform for pervasive health care monitoring. *Appl. Soft Comput.* **2011**, *11*, 315–325. [[CrossRef](#)]
30. Vitabile, S.; Conti, V.; Militello, C.; Sorbello, F. An extended JADE-S based framework for developing secure Multi-Agent Systems. *Comput. Stand. Interfaces* **2009**, *31*, 913–930. [[CrossRef](#)]
31. Agirre, A.; Parra, J.; Armentia, A.; Ghoneim, A.; Estévez, E.; Marcos, M. QoS management for dependable sensory environments. *Multimed. Tools Appl.* **2015**. [[CrossRef](#)]
32. OSGiTM Alliance. The OSGi Architecture. Available online: <http://www.osgi.org/Technology/WhatIsOSGi> (accessed on 22 October 2015).
33. Bloomer, J. *Power Programming with RPC*; O'Reilly Media: Sebastopol, CA, USA, 1992.
34. Object Management Group. Object Request Broker (ORB). Available online: http://www.omg.org/gettingstarted/orb_basics.htm (accessed on 22 October 2015).
35. Capra, L.; Emmerich, W.; Mascolo, C. CARISMA: Context-Aware Reflective middleware System for Mobile Applications. *IEEE Trans. Softw. Eng.* **2003**, *29*, 929–945. [[CrossRef](#)]
36. Foundation for Intelligent Physical Agents. Standard FIPA Specifications. Available online: <http://www.fipa.org/repository/standardspecs.html> (accessed on 7 September 2015).
37. Krupitzer, C.; Roth, F.M.; VanSyckel, S.; Schiele, G.; Becker, C. A survey on engineering approaches for self-adaptive systems. *Pervasive Mob. Comput.* **2014**, *17*, 184–206. [[CrossRef](#)]
38. Armentia, A.; Agirre, A.; Estévez, E.; Pérez, J.; Marcos, M. Model Driven Design Support for Mixed-Criticality Distributed Systems. In Proceedings of the 19th World Congress of the International Federation of Automatic Control, Cape Town, South Africa, 24–29 August 2014; pp. 4441–4446.
39. Armentia, A.; Sarachaga, I.; de Albeniz, O.G.; Estevez, E.; Aguirre, A.; Marcos, M. Achieving Reconfigurable Service Oriented Applications Using Model Driven Engineering. In Proceedings of the 16th IEEE Conference on Emerging Technologies & Factory Automation, Toulouse, France, 5–9 September 2011; pp. 1–4.
40. Armentia, A.; Gangoiti, U.; Priego, R.; Marcos, M. A Multi-Agent Based Approach to Support Adaptability in Home Care Applications. In Proceedings of the 2nd Conference on Embedded Systems, Computational Intelligence and Telematics in Control, Maribor, Slovenia, 22–24 June 2015; pp. 1–6.
41. Farella, E.; Falavigna, M.; Ricc, B. Aware and smart environments: The Casattenta project. *Microelectron. J.* **2010**, *41*, 697–702. [[CrossRef](#)]

42. Søberg, J.; Goebel, V.; Plagemann, T. CommonSense: Personalisation of Complex Event Processing in Automated Homecare. In Proceedings of the 6th International Conference on Intelligent Sensors, Sensor Networks and Information Processing, Brisbane, Australia, 7–10 December 2010; pp. 275–280.
43. Botia, J.A.; Villa, A.; Palma, J. Ambient Assisted Living system for in-home monitoring of healthy independent elders. *Expert Syst. Appl.* **2012**, *39*, 8136–8148. [[CrossRef](#)]
44. Benghazi, K.; Hurtado, M.V.; Hornos, M.J.; Rodríguez, M.L.; Rodríguez-Domínguez, C.; Pelegrina, A.B.; Rodríguez-Fórtiz, M.J. Enabling correct design and formal analysis of Ambient Assisted Living systems. *J. Syst. Softw.* **2012**, *85*, 498–510. [[CrossRef](#)]
45. Rabbi, F.; Lamo, Y.; Maccaull, W. A Flexible Metamodelling Approach for Healthcare Systems. In Proceedings of the 2nd European Workshop on Practical Aspects of Health Informatics, Trondheim, Norway, 19–20 May 2014; pp. 115–128.
46. Rocha, A.; Martins, A.; Freire, J.C.; Kamel Boulos, M.N.; Vicente, M.E.; Feld, R.; van de Ven, P.; Nelson, J.; Bourke, A.; ÓLaighin, G.; et al. Innovations in health care services: The CAALYX system. *Int. J. Med. Inf.* **2013**, *82*, e307–e320. [[CrossRef](#)] [[PubMed](#)]
47. Perry, M.; Dowdall, A.; Lines, L.; Hone, K. Multimodal and ubiquitous computing systems: Supporting independent-living older users. *IEEE Trans. Inf. Technol. Biomed.* **2004**, *8*, 258–270. [[CrossRef](#)] [[PubMed](#)]
48. Ballagny, C.; Hameurlain, N.; Barbier, F. MOCAS: A State-Based Component Model for Self-Adaptation. In Proceedings of the 3rd IEEE International Conference on Self-Adaptive and Self-Organizing Systems, San Francisco, CA, USA, 14–18 September 2009; pp. 206–215.
49. Sadri, F. Ambient intelligence. *ACM Comput. Surv.* **2011**, *43*, 1–66. [[CrossRef](#)]
50. Selic, B. The pragmatics of model-driven development. *IEEE Softw.* **2003**, *20*, 19–25. [[CrossRef](#)]
51. Duran-Limon, H.A.; Blair, G.S.; Friday, A.; Grace, P.; Samartzidis, G.; Sirvaharan, T.; Wu, M. *Context-Aware Middleware for Pervasive and Ad Hoc Environments*; Technical Report; Computing Department, Lancaster University: Lancaster, UK, 2003.
52. Khan, M.U.; Reichle, R.; Geihs, K. Architectural constraints in the model-driven development of self-adaptive applications. *IEEE Distrib. Syst.* **2008**, *9*, 1–10. [[CrossRef](#)]
53. Morin, B.; Barais, O.; Jezequel, J.M.; Fleurey, F.; Solberg, A. Models@ Run.time to Support Dynamic Adaptation. *Computer* **2009**, *42*, 44–51. [[CrossRef](#)]
54. Anthony, R.; Rettberg, A.; Chen, D.; Jahnich, I.; de Boer, G.; Ekelin, C. Towards a Dynamically Reconfigurable Automotive Control System Architecture. In Proceedings of the Working Conference: International Embedded Systems Symposium, Irvine, CA, USA, 30 May–1 June 2007; pp. 71–84.
55. García-Valls, M.; Rodríguez López, I.; Fernández Villar, L. iLAND: An Enhanced Middleware for Real-Time Reconfiguration of Service Oriented Distributed Real-Time Systems. *IEEE Trans. Ind. Informatics* **2011**, *9*, 228–236. [[CrossRef](#)]
56. Kumar, M.; Shirazi, B.A.; Das, S.K.; Sung, B.Y.; Levine, D. PICO: A Middleware Framework for Pervasive Computing. *IEEE Pervasive Comput.* **2003**, *2*, 72–79. [[CrossRef](#)]
57. Gharzouli, M.; Boufaida, M. A generic P2P Collaborative Strategy for Discovering and Composing Semantic Web Services. In Proceedings of the 4th International Conference on Internet Web Applications and Services, Venice/Mestre, Italy, 24–28 May 2009; pp. 449–454.
58. Eichelberg, M.; Rein, A.; Blisching, F.; Wolf, L. The GAL Middleware Platform for AAL: A Case Study. In Proceedings of the first International Workshop on AAL Service Platforms, Lyon, France, 2 July 2010; pp. 1–6.
59. Chen, C.M. Web-based remote human pulse monitoring system with intelligent data analysis for home health care. *Expert Syst. Appl.* **2011**, *38*, 2011–2019. [[CrossRef](#)]
60. Witting, M.D.; Lueck, C.H. The ability of pulse oximetry to screen for hypoxemia and hypercapnia in patients breathing room air. *J. Emerg. Med.* **2001**, *20*, 341–348. [[CrossRef](#)]
61. Holborn, P.; Nolan, P.; Golt, J. An analysis of fatal unintentional dwelling fires investigated by London Fire Brigade between 1996 and 2000. *Fire Saf. J.* **2003**, *38*, 1–42. [[CrossRef](#)]
62. Jobbágy, Á.; Csordás, P.; Mersich, A. Blood Pressure Measurement at Home. In Proceedings of the 2006 World Congress on Medical Physics and Biomedical Engineering, Seoul, Korea, 27 August–1 September 2006; pp. 3453–3456.

63. Hervás, R.; Fontecha, J.; Ausín, D.; Castanedo, F.; Bravo, J.; López-de-Ipiña, D. Mobile monitoring and reasoning methods to prevent cardiovascular diseases. *Sensors* **2013**, *13*, 6524–6541. [[CrossRef](#)] [[PubMed](#)]
64. Bellifemine, F.; Caire, G.; Poggi, A.; Rimassa, G. JADE: A software framework for developing multi-agent applications. Lessons learned. *Inf. Softw. Technol.* **2008**, *50*, 10–21. [[CrossRef](#)]
65. Baldauf, M.; Dustdar, S.; Rosenberg, F. A survey on context-aware systems. *Int. J. Ad Hoc Ubiquitous Comput.* **2007**, *2*, 263–277. [[CrossRef](#)]
66. Cooking hacks. e-Health Sensor Platform V2.0 for Arduino and Raspberry Pi [Biometric / Medical Applications]. Available online: <https://www.cooking-hacks.com/documentation/tutorials/ehealth-biometric-sensor-platform-arduino-raspberry-pi-medical> (accessed on 13 October 2015).
67. Cooking hacks. Waspote Gas Sensors Kit. Available online: <https://www.cooking-hacks.com/shop/waspote/kits/waspote-gas-sensors-kit> (accessed on 13 October 2015).
68. Siegel, E.; Retter, A. *eXist: A NoSQL Document Database and Application Platform*; O'Reilly Media: Sebastopol, CA, USA, 2014.



© 2015 by the authors; licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons by Attribution (CC-BY) license (<http://creativecommons.org/licenses/by/4.0/>).

3.2 Model-Driven Design and Development of Flexible Automated Production Control Configurations for Industry 4.0

Gangoiti, U., López, A., Armentia, A., Estévez, E., y Marcos, M. (2021). Model-Driven Design and Development of Flexible Automated Production Control Configurations for Industry 4.0. *Applied Sciences*, 11 (5), 2319, pp. 1–27.

DOI: <https://doi.org/10.3390/app11052319>.

JCR©2021: 2,838

Categoría: Engineering, Multidisciplinary

Cuartil: Q2 (39/92)

Article

Model-Driven Design and Development of Flexible Automated Production Control Configurations for Industry 4.0

Unai Gangoiti ¹, Alejandro López ¹ , Aintzane Armentia ¹ , Elisabet Estévez ^{2,*}  and Marga Marcos ¹ 

¹ Automatic Control and Systems Engineering Department, University of the Basque Country, 48013 Bilbao, Spain; unai.gangoiti@ehu.eus (U.G.); alejandro.lopez@ehu.eus (A.L.); aintzane.armentia@ehu.eus (A.A.); marga.marcos@ehu.eus (M.M.)

² Electronic and Automation Engineering Department, University of Jaén, 23071 Jaén, Spain

* Correspondence: eestevez@ujaen.es; Tel.: +34-95-321-2167

Abstract: The continuous changes of the market and customer demands have forced modern automation systems to provide stricter Quality of service (QoS) requirements. This work is centered in automation production system flexibility, understood as the ability to shift from one controller configuration to a different one, in the most quick and cost-effective way, without disrupting its normal operation. In the manufacturing field, this allows to deal with non-functional requirements such as assuring control system availability or workload balancing, even in the case of failure of a machine, components, network or controllers. Concretely, this work focuses on flexible applications at production level, using Programmable Logic Controllers (PLCs) as primary controllers. The reconfiguration of the control system is not always possible as it depends on the process state. Thus, an analysis of the system state is necessary to make a decision. In this sense, architectures based on industrial Multi Agent Systems (MAS) have been used to provide this support at runtime. Additionally, the introduction of these mechanisms makes the design and the implementation of the control system more complex. This work aims at supporting the design and development of such flexible automation production systems, through the proposed model-based framework. The framework consists of a set of tools that, based on models, automate the generation of control code extensions that add flexibility to the automation production system, according to industry 4.0 paradigm.

Keywords: flexible automation production systems; model driven engineering; multi agent system; I4.0 components



Citation: Gangoiti, U.; López, A.; Armentia, A.; Estévez, E.; Marcos, M. Model-Driven Design and Development of Flexible Automated Production Control Configurations for Industry 4.0. *Appl. Sci.* **2021**, *11*, 2319. <https://doi.org/10.3390/app11052319>

Academic Editor: Yaniv Mordecai

Received: 13 January 2021

Accepted: 2 March 2021

Published: 5 March 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

In the last years, there is an increasing interest in making manufacturing systems more competitive. Some countries use different terms to refer to this phenomenon, for example it is known as Advanced Manufacturing in U.S., Industrie 4.0 in Germany and Factory of the Future in other European Countries [1–3]. Basically, this evolution consists in integrating all production systems to pass from long batches, which seek costs reduction through scale economies, to a flexible and personalized production [4]. In other words, all these initiatives have a common goal: achieving high quality production with zero defects [5,6]. For this, they base on the so-called smart factory, composed by adaptive and smart manufacturing equipment and systems, which enables the automation, control and optimization of high-tech manufacturing processes while assuring the availability of the plant. The smart factories control all their processes, but at the same time, they should also be connected to the market, the supply, and the demand. This paradigm, generally referred as Industry 4.0, has countless applications both in academia and in Industry [7–9] since the birth of the term in 2011 [10,11].

Dynamic reconfiguration is being adopted by current automation systems in order to ensure Quality of Service (QoS) requirements. In a production system the QoS requirements (also called non-functional requirements) can be regarded as those properties that improve

product attractiveness, usability, accuracy, safety or reliability without modifying product functionality. Non-functional requirements demand specific behavior to the manufacturing system such as, for instance, reliability, availability, power consumption or response time optimization. In particular, the experience of large industrial companies has shown that two of the main qualities of advanced manufacturing systems are: *flexibility* and *adaptability*, which characterize systems with the ability to quickly adapt to environment [12]. In general, all these reconfiguration mechanisms enable to switch in the most quick and cost-effective way from one configuration to another at runtime. As a result, the system response to sudden changes on customer demands or even to unpredictable events (e.g., failures or disruptions) is improved. This work is centered in automation production system flexibility, understood as the way to deal with non-functional requirements such as assuring control system availability under failure of a machine, components, network or controllers.

According to different reviews and surveys [13–15] the term ‘reconfiguration’ may make reference to: (1) product reconfiguration, as the flexibility to change or modify the final product. (2) Schedule reconfiguration, which is commonly understood as the capability to modify the execution order of plant operation, for enhancing efficiency or productivity [16,17] or overcoming machine failures [18]. (3) Sometimes it also refers to machine operation reconfiguration. That is, modifying the functionality of a machine to enable it to perform other operations [12,19,20]. Finally, (4) control system reconfiguration refers to the relocation of the different functionalities within a distributed control system, for improving the controller performance [21] or battery consumption [22], or even avoiding service disruption in case of failures at controller or network [23–25].

These works provide a custom solution as they are focused on assuring a specific QoS (e.g., optimization of the production, fault tolerance at process or controller level and workload balance). Additionally, the majority make use of programming languages, like C/C++ or Java, which are not widely used in the factory automation area. Therefore, they are ad hoc solutions and/or they are not easily adopted in industrial environments.

The goal of this work is twofold. On the one hand, it proposes a generic implementation of reconfigurable automation applications to be executed under the control of the so-called Flexible Automation Middleware (FAM) presented in [26]. This is a generic Multi-Agent System (MAS) that can be particularized for the supervision of a concrete set of system QoS, launching a system reconfiguration in case of QoS loss. On the other hand, it presents a flexible model-based framework, which, based on well-spread and accepted standards, helps the designer to define the information needed to achieve dynamic reconfiguration of the automation system.

Hence, this paper contributes: (1) A modeling approach that collects information about the production process and the distributed automation system, which is relevant for the management platform that makes use of it. (2) Application templates for the runtime agent-based platform. (3) A tool suite that implements the approach, aimed at adding flexibility to the original distributed automation system, supporting dynamic reconfiguration of the control system due to controller’s fault or work balance.

The remainder of the paper is as follows: Section 2 details the meaning of Flexible Automation Production Systems and a brief description of the FAM agent-based architecture. This section also justifies why Model Driven Engineering (MDE) is useful for designing and developing complex automation systems. Section 3 presents a framework that has as main goal the automatic generation of the so-called flexible automation projects. Additionally, this framework also gives support to the automatic generation of the sets of agents that are application. Section 4 is devoted to assessing the system flexibility through a case study. Finally, Section 5 outlines the most important conclusions and future works.

2. Materials and Methods

The Industry 4.0 paradigm frames the technologies and conventions required to achieve the reconfiguration of the control systems. Based on Industry 4.0 principles, specifically in the Reference Architecture Model for Industry 4.0 (RAMI 4.0) [27], first

subsection characterizes the Flexible Automation Production System (FAPS) in which reconfiguration can take place, switching from one controller configuration to a different one, in the quickest and cost-effective way, without disrupting its normal operation.

The implementation of FAPS is a complex task, and requires analysis, decision-making and negotiation abilities, which can be achieved by means of agent-based solutions. In this sense, an architecture based on industrial Multi Agent Systems can provide this support at runtime. In the literature, there are some MA-based approaches for manufacturing, e.g., [25,28,29] which agents can be deployed in some very different devices, providing support to a certain QoS parameters. Nevertheless, as far as authors know, only the agent-based architecture proposed in [26] supports the controller fault, launching a reconfiguration of the control system when this QoS loss is detected. The main ideas of this Flexible Automation Middleware (FAM) are presented in Section 2.2.

Furthermore, in order to enable reconfiguration of the control systems assuring normal operation to continue, it is required to model the possible states of the manufacturing process. Thus, Section 2.3 discusses how MDE can be a helpful alternative for modeling the industrial agents of the FAPS in such way.

2.1. Modeling of Flexible Automation Production Systems

Reference Architecture Model for Industry 4.0 (RAMI 4.0) [27] offers a structured description of the fundamental requirements of I4.0-compliant systems, exploring: (1) hierarchical levels of a manufacturing system networked via the Internet; (2) the lifecycle of systems and products; and (3) the Information Technology management layers of I4.0 project implementation. In RAMI 4.0, any technical asset of the factory has a digital representation as an I4.0 component, which are provided with digital interfaces to interact with other I4.0 components. Hence, the I4.0 component is the combination of objects from both the physical world and the information world, offering dedicated functionalities and flexible services to other I4.0 Components [30]. As stated in [27], I4.0 components have two main features:

1. The *Asset* is a physical or a logical object owned by or under the custodial duties of an organization, having either a perceived or actual value to the organization.
2. The *Asset Administration Shell* (AAS) is the digitalization of an asset. In other words, and AAS is the interface that connects the physical asset through I4.0 communications such as OPC Unified Architecture (OPC UA) [31]. It is also in charge of offering the I4.0 component's services to the Industry 4.0 (e.g., [32] presents an AAS model able to represent International Electrotechnical Commission (IEC) 61131-3 standard compliant programs and the relevant relationships with Programmable Logic Controllers and each device of the controlled plant). According to the glossary of Platform Industrie 4.0 [33], the AAS concept can be directly related to the concept of Digital Twin (DT). However, several works in the literature apply the DT concept to refer exclusively to highly accurate simulation models. In the authors' understanding, both meanings are correct as both, in different ways, are related to the Asset behavior. In this work, the first one applies.

Following this concept, Automation Production Systems (APSs) could be viewed as a set of two types of I4.0 connected Components: (1) Controller and (2) Plant, which offer production services. Their main features are the following:

- Controller I4.0 Component:
 - *Asset (ControllerAsset)*: The Programmable Logic Controller (PLC), the primary controller for such type of systems, and the I/O boards;
 - *AAS (AAS_Controller)*: automatizes the production by means of its I/Os.
- Plant I4.0 Component:
 - *Asset (PlantAsset)*: physical station and the sensors and actuators connected to controller's I/Os;

- AAS (*AAS_Plant*): offers a production service (which contains the control logic and data).

This work goes a step further, and it defines Flexible Automation Production Systems (FAPSs) as those APSs that support flexibility through reconfiguration according to QoS parameters, such as control system availability in the event of controller failure or workload balancing. Hence, this work provides support for developing flexible applications at production level, using PLCs as primary controllers. To achieve this, during the design of the FAPSs it must be stated which controllers can potentially automate the production of every *PlantAsset* in order to download the corresponding Control Logic Software Module (a Program Organization Unit—POU- if IEC 61131-3 standard [34] is used). The runtime platform manages the execution of every control software module (henceforth Production Service-PS-), ensuring that it is only running in a unique controller of the system (active controller), and the others acting as tracking controllers. Besides, applying dynamic reconfiguration to assure specific QoS (e.g., availability of the control system, efficient use of resources or any other QoS), implies both QoS supervision (by means of mechanisms like heartbeat or workload, respectively, in all controllers) and guaranteeing reconfiguration feasibility, avoiding unpredictable effects in the manufacturing process. Therefore, during the design phase, it is also necessary to define those critical situations in which the state of the *PlantAsset* is not known and thus, the reconfiguration is not possible.

Figure 1 depicts the general scenario of a FAPS in terms of I4.0 Components. This seeks to prove the capability to assure work balance among the distributed controllers (*AAS_Controller1* . . . *AAS_Controller3*) comprising the automation system for a flexible manufacturing cell which is composed by three stations (*PlantAsset_S1* . . . *PlantAsset_S3*). As observed, *AAS_Controllers*, as composed AASs, contain a set of *AAS_Plants*. In this example, the software control code of S1 (*AASPlant_S1*) can be run in either controller 1 or controller 2, but it is only active in controller 1.

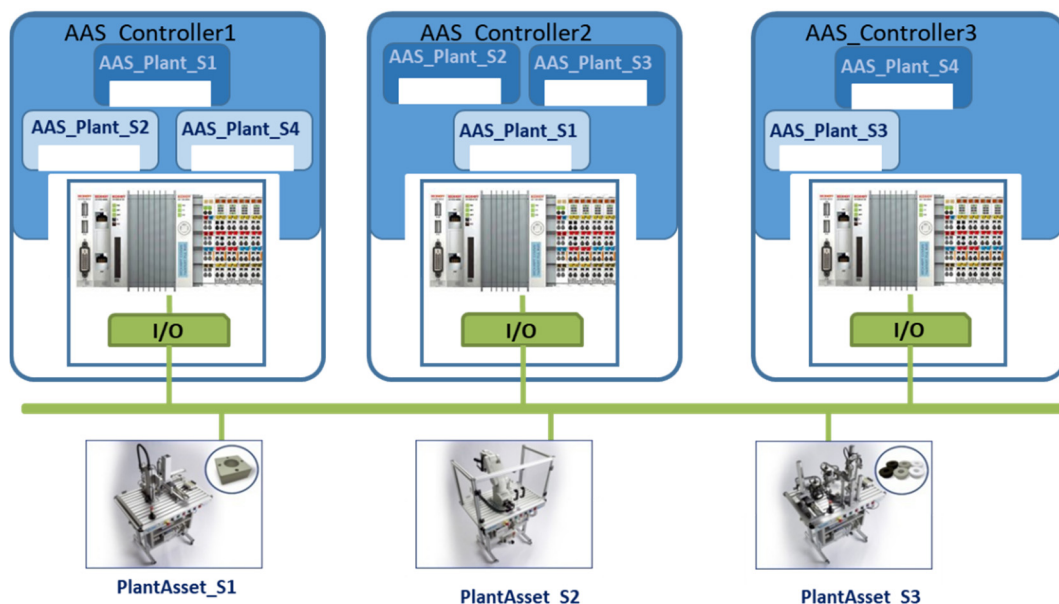


Figure 1. A Flexible Automation Production System with the following I4.0 Components: multiple Asset Administration Shell (AAS) of different assets (Controllers and Plant) are interconnected via a message exchange middleware, depicted as a message bus.

The structure of AASs is defined in [35]. Following this recommendation, Figure 2. illustrates the general structure proposed for the *AAS_Controller*, which has two main parts: Header and Body. The Body has two submodels per *AAS_Plant* contained (Production-Service and ProductionService Availability) and the *ActiveProductionServices* submodel

that collects the list of ProductionServices which are being executed in the corresponding ControllerAsset.

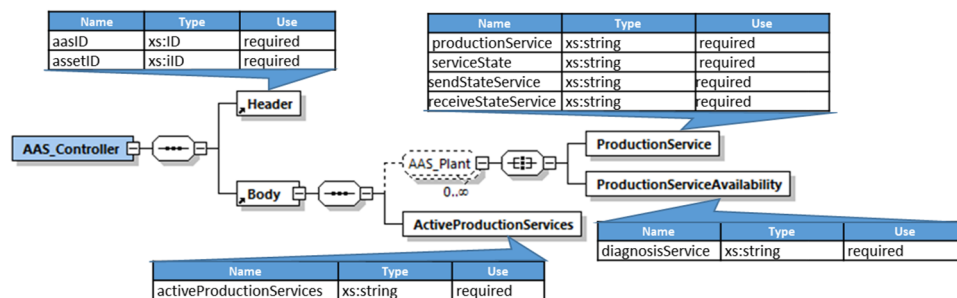


Figure 2. Structure of a generic AAS_Controller: the figure follows the concept of an AAS from Platform Industrie 4.0 [35] to define the AAS of a controller implemented as a Markup Language. Three Sub-Models are defined, the first two related to the set of Plant Assets it can control (AAS_Plant) and the third that informs about the Plant Assets it actually controls. Every sub-model has properties that specify the actions and/or information it provides. This graphic was generated using Altova XMLSpy Version 2020.sp1.

Production Service submodels are characterized by four properties: The service offered by the corresponding *AAS_Plant* (ProductionService); a set of variables that characterize the state of such production service (Service State); and two functions to send or receive the values of the variables that collect the execution state. Active controllers send at the end of the execution cycle the current execution state (SendStateService) and those controllers which are tracking receive this state (ReceiveStateService). Hence, if an active controller fails all tracking controllers have the last known state.

Production Service Availability submodels have a unique property for runtime diagnosis. Thus, this function indicates if the current state of the *PlantAsset* can be derived from the current values of the controller variables or not. To perform such diagnosis this function processes the set of critical situations detailed during the design phase. At runtime, a *PlantAsset* could be in the following states or situations:

- *non-critical* situations: the automation system is aware of the current process state. Therefore, it is possible to activate ProductionServices in another PLC (after de-activation in the current controller or after a controller failure), using as their initial state the last known state of the interrupted ProductionService;
- *critical* situations: the automation system does not know exactly the current state of the ProductionServices. Therefore, as it can lead to unpredictable process behavior, the activation/de-activation of ProductionServices is inhibited in critical situations. For example, when a controller fails, it has to be analyzed if all its active ProductionServices can be recovered, on a tracking controller, in a previous known state (checkpoint). In the case of a non-recoverable situation, it is analyzed if the ProductionService must be safely stopped and the operator warned. Figure 3 depicts a simple but illustrative example of a ProductionService for the movement of a piece by a crane. While the crane is lifting, transporting or placing the piece, the ProductionService cannot be deactivated as the piece may be released which prevents the production from continuing. Therefore, the state of the ProductionService during these operations is denoted as critical and in such case, reconfiguration cannot be performed. The rest of the states are denoted as non-critical.

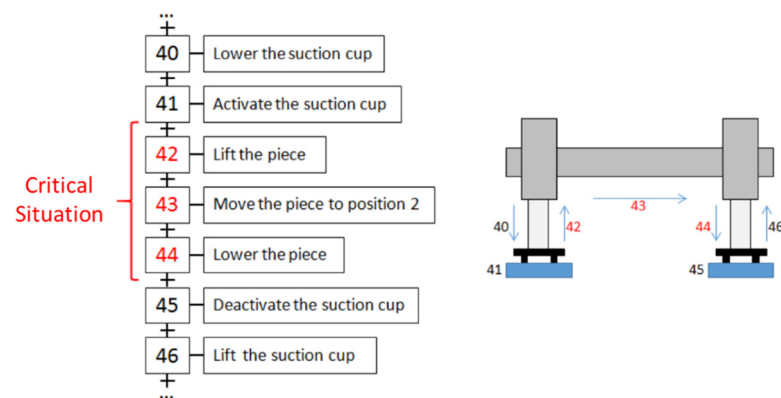


Figure 3. Example of ProductionService to illustrate the concept of critical situation: the figure depicts the manufacturing sequence of a crane. The steps 42 to 44 are critical situations because the state of the system is uncertain during their execution (in case of failure there is no information about the exact position of the piece). Therefore, reconfiguration cannot be performed during such steps.

2.2. Flexible Automation Middleware (FAM)

This subsection summarizes the agent-based middleware architecture for flexible automation production systems proposed by the authors in [26]. The general scenario is illustrated in Figure 4.

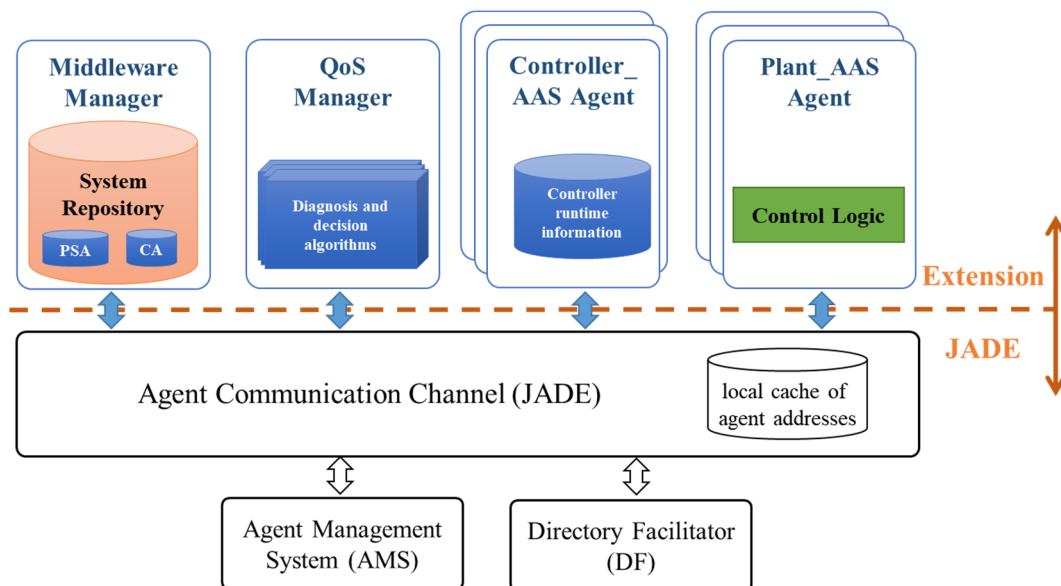


Figure 4. General Scenario of the FAM (customization of [26]): the Middleware Manager and the QoS Manager (comprising the QoS Monitor Agent and the Diagnosis & Decision Agent) constitute the core of the middleware. The former is in charge of managing the execution and maintaining the state of Controllers and Plant AAS Agents. The latter is in charge of supervising the availability of the overall control system, detecting controller faults and, if possible, recovering the Plant AASs of the failed controller.

FAM includes four agent types: two are part of the generic and basic architecture, whereas the other two are application dependent. Generic agents are able of managing different QoS:

1. The *Middleware Manager (MM)* is the main orchestrator. It is a unique agent in charge of managing the *System Repository (SR)*: a dynamic model that contains information about the current state of the automation application, which changes over time.
2. The *QoS Manager* comprises a set of agents responsible for QoS fulfilment.

- a. *QoS Monitor (QM)* agents are responsible for monitoring the specific QoS to be handled, generating triggers if they detect QoS losses. Hence, there are as many QMs as QoS to be met.
- b. *The Diagnosis & Decision (D&D)* agent is unique in the system and it is responsible for launching diagnosis and decision algorithms as well as reconfiguration events.

As commented above, the rest of agents in the system depend on the automation application. Applied to this work they implement both *Plant_AAS* and *Controller_AAS* assuring the availability. These agents guarantee the distributed intelligence, as their role is to collect information from the current state of the automation system as well as performing reconfiguration decided by the D&D:

3. The *Plant_AAS Agent* (APlant_AAS) manages the execution of the corresponding ProductionService's actions as well as collects, transmits, stores and makes diagnosis on the current state of this. The Component Manager of a APlant_AAS is implemented by a Finite State Machine that represents the possible states of the ProductionService lifecycle (detailed description can be found in [26]). Although each ProductionService can be replicated in a number of controllers, at runtime only one controller will be executing the ProductionService and its corresponding APlant_AAS will be in active state. For the rest of the controllers, the ProductionService is not executing and their corresponding APlant_AASs will be in tracking state.
4. The *Controller_AAS Agent* (AController_AAS) registers its corresponding controller and the associated resources in the System Repository when the controller joins the system. It also registers itself in the Directory Facilitator of JADE offering as services the set of ProductionServices that can run in the controller. Finally, it launches its corresponding APlant_AASs. There are as many AController_AAS as controllers in the system.

2.3. Model Driven Engineering for Modeling Flexible Automation Production Systems

This subsection illustrates how Model Driven Design has been adopted for designing and developing automation systems. In fact, it has been used for both characterization purposes (from the definition of system parts, such as QoS requirements, to the overall system description) and implementation purposes. In the concrete case of industrial automation field, model-based techniques are integrated into the development process. Several works base on the use of the Unified Modeling Language (UML [36]) to describe control systems based on the IEC 61131 [37,38] and the IEC 61499 [39,40] standards. The Systems Modeling Language (SysML [41]) has been also applied [39,42], whereas other works also use modeling techniques and design patterns [43] or aspects [44]. Furthermore, the worldwide PLCopen association, which is vendor and product-independent, has specified a common representation format for the software model of the IEC 61131-3 standard [45,46]. The objective is twofold. On the one hand, it is aimed at achieving programming tools interoperability. On the other hand, it also supports a model-based definition of the application software of automation systems.

Other authors go a step further and make use of modeling techniques for supporting the development of the overall automation system. The "3 + 1" architecture proposed by Thramboulidis [47] allows the system design based on three models (software engineering, mechanical engineering and electrical engineering) linked through the "+1" model, which in the end conforms the whole system. Another example is the MDD approach proposed in [37,48], which uses the UML profile technique to define domain languages. Additionally, it also allows the automatic generation of the software architecture in PLCopen XML format, using functional code imported from PLC libraries. Similarly, authors in [49] define the so-called SysML-AT, a specialized profile that is integrated into the German commercial tool CoDeSys, and that allows the definition of the hardware and software of the automation and control systems.

Models have been also used to consider system reconfiguration as an extension of the definition of system elements, such as Function Blocks (FB), machines, controllers or components. This is the case of the Functional Application Design for Distributed Automation Systems (FAVA) research project [43,50], which proposes including resource demands within the software view (amount of memory and number of bytes exchanged with other FBs) with the aim to be used at the deployment of the FBs.

The model-based approach presented in [51] considers both functional and non-functional requirements in terms of constraints related to the different views of the production automation system. It covers from sensors or actuators to the whole plant, including a tolerance model with traceability purposes. In fact, the information contained in this model is used by an agent system for analyzing if reliability demands can be maintained, and whether the needed probability of a concrete quality will be reached. Non-functional requirements are also tackled at the AMoDE-RT approach [44], but as an aspect-based characterization related to the functional components. As a result, non-functional demands can be supervised at runtime, being possible to reconfigure the system in the event of non-fulfillment. The approach is applied in [42] to embedded control systems.

The holonic architecture at the SOCRADES project [52] performs runtime distribution of machine job, through a model-based definition of the functionalities of a machine. The modeling approach described in [53] allows the specification of the operations performed by the machines within a plant as well as the operations required for a product manufacturing. This information is used to automatically derive an optimal operation sequence.

All research works commented above demonstrate the usefulness of MDE in automation field for different purposes, as all of them have in common that the use of models helps managing the complex automation systems [54,55]. This work uses MDE in order to design and develop Flexible Automation Production Systems, which are complex systems due to their size, functionality and distribution. In fact, MDE relies on models and model transformations for automating the software development process. More precisely, there are Model to Model (M2M) transformations as well as Model to Text (M2T) transformations. In both cases, the input refers to a model that conforms to a meta-model whereas the generated output is related to a new model conforming to another meta-model or source code, respectively.

The authors propose the Meta-Model illustrated in Figure 5, which collects all the information for defining FAPSs. The framework proposed by the authors implements this Meta-Model in a Markup Language file. This model is the input of the: (1) M2M transformation rules to generate as many flexible automation projects as controllers in the system, in PLCopen XML format; (2) M2T transformation rules to generate the code of APlant_AASs and AController_AASs in PLCopen XML format. The Technical Committee six of PLCopen defines a meta-model in a ML notation (XML Schema) for the IEC 61131-3 standard software model. This ensures that the M2M transformation produces models following the PLCopen meta-model.

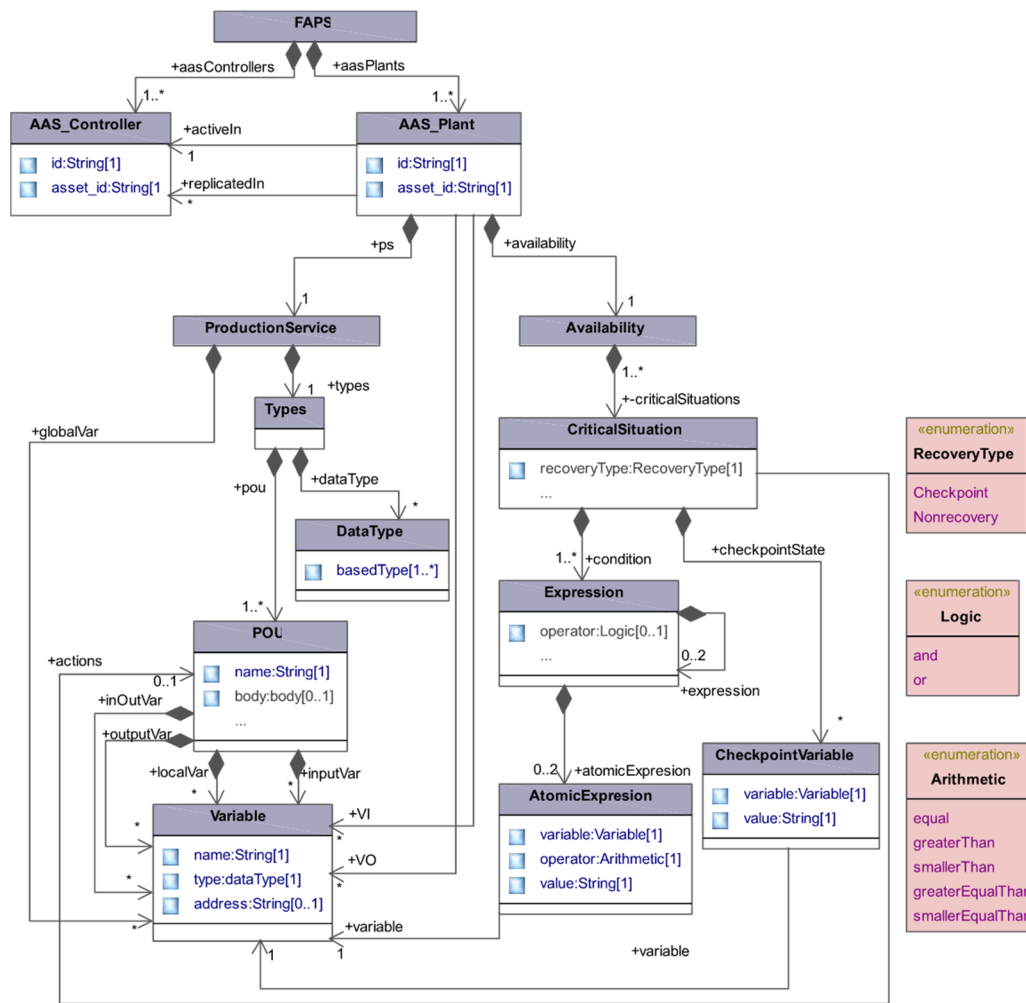


Figure 5. Flexible Automation Production System Meta-Model: this Meta-Model represents both, the formal expression of the software in charge of controlling a Plant Asset (on the left side) as well as the information needed in order to analyze if the control of a Plant Asset might be recoverable under a controller fault/workload balancing (on the right side). The corresponding code resides in every controller which potentially can control the Plant Asset.

3. Results

This section presents a MDE based framework that generates: (1) a flexible automation project per PLC of the FAPS; and (2) a set of application dependent agents (APlant_AASs and AController_AASs). Additionally, in order to make FAM generic and customizable, this work defines the templates for these types of agents which are customized with application dependent information. As commented above, these results are achieved applying a set of M2M and M2T transformation rules to a Model that must be specified since the design phase following the meta-model depicted in Figure 5.

The general scenario of the proposed framework is illustrated in Figure 6. It is based on two automation standards: PLCopen [46] and AutomationML [56,57]. FAM is composed of two core elements: (1) The FAPS Model Editor, and (2) the code generator. As previously commented, two different outcomes are obtained from code generation: on the one hand, the Flexible Automation Projects, which are composed by the set of ProductionServices the PLC can run, as well as the activation/de-activation code and the recovery actions; on the other hand, the code corresponding to the application agents.

The following subsections detail the FAPS Model Editor as well as the code generators, which are based upon M2M and M2T transformations, respectively. As the input for these transformations (i.e., the outcome from the FAPS Model Editor) is in XML, the identified transformation rules will be implemented by using XML stylesheet technology [58].

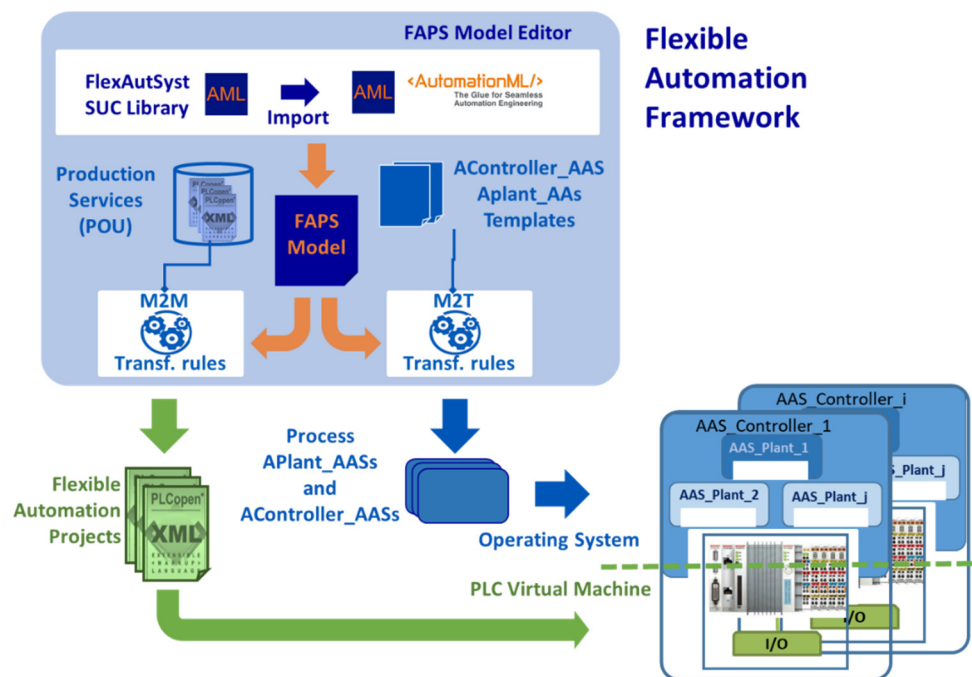


Figure 6. General Scenario of Flexible Automation Framework.

3.1. FAPS Model Editor

The design and development of the automation system must be structured in Production Services, which, by means of I/Os and their control logic as a set of POU and variables, control different stages of the process. The POU of the control logic can be generated directly in a PLC programming tool or following the guidelines provided in [48] or [49]. ProductionServices are duplicated in different controllers (*replicatedIn* in Figure 5). Moreover, each *AAS_Plant* is characterized by a set of critical situations that refer to situations at which ProductionService cannot be reconfigured. Critical situations are defined by a condition to be met, which is defined as logical expressions of ProductionService variables. For instance, in the event of a controller failure, the continuity of the automation system execution must be analyzed, resulting in two alternative outcomes: (1) the normal execution can be restored by means of recovery actions, taking the system to a known previous state (checkpoint); (2) the failure is not recoverable, and thus a safe stop action is required.

The FAPS Model Editor provides support to developers for designing automation systems. This tool follows the guidelines of [59] to implement the meta-model of Figure 5 using the Computer Aided Engineering eXchange (CAEX) [60] libraries of AutomationML:

- The System Unit Class Library indicates the concepts required to define a flexible automation system. It comprises the so-called System Unit Classes (SUC), which represent the elements of the meta-model. The SUCs are characterized by their attributes. As the elements in the system can be simple or complex (i.e., composed of internal elements), the SUCs representing them can be either simple or complex. The complex SUCs comprise instances of other previously defined SUCs.
- The Interface Class Library offers interfaces that enable the association of a SUC (simple or complex) to an element on an external file. This library provides a PLCopen interface included in AML which grants access to the POU and variables within a PLCopen automation project. It also includes the hardware interface, a new interface added to allow the definition of the controllers and automation projects in PLCopen.
- The Role Class Library provides the different roles by which the elements can be organized in the model (choice, sequence, each and every role). The ramifications of

the structures based on these roles is settled by means of their attributes minOccurs and maxOccurs.

Further details regarding these libraries and their specifications can be checked in [59].

These libraries are integrated into the AML editor (see *FlexibleAutomationSystem* SUC library in Figure 7). This allows to use this tool to define FAPS models. Every Internal Element (IE in Figure 7) is an object whose Class corresponds to another SUC of the library. The basic attributes of the *AAS_Plant* are its id, asset_id and the reference to the controller in which the control logic is active (activeIn) as well as the latent controllers (replicatedIn). This definition is completed with links to POUs and global variables. This latter is a sequence of critical situations which are defined making use of expressions involving variables.

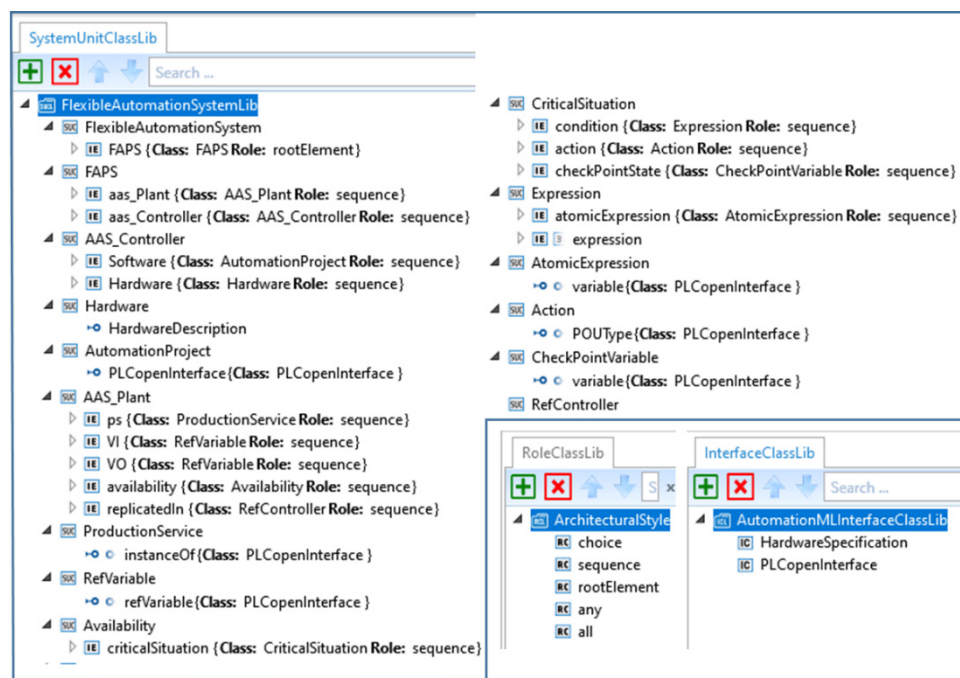


Figure 7. CAEX libraries for Flexible Automation Systems: excerpt of a SystemUnitClassLib with AutomationML Editor. It comprises multiple System Unit Class (SUC) to define the lexicon participating in the definition of a Flexible Automation System (FAPS, AAS_Controllers, Hardware ...).

As an example, the definition of the *AAS_Plant* named ST1 is depicted in Figure 8.

Besides, the FAPS Model Editor allows the characterization of the critical situations of PSs. To that end, the developers must conform boolean expressions to evaluate check-point or unrecoverable critical situations from arithmetic and logical operations with the available variables. In the same way, the recovery actions to be performed at each checkpoint state (i.e., the values of the variables that define the checkpoint state), can be declared at this point, if needed. Figure 8 presents the definition of the expression: “(Control_ST1.Sequence1_1.E23.Q1 = 1 AND Control_ST1.Insert_P1.E73.Q1 = 1)” using the AML editor.

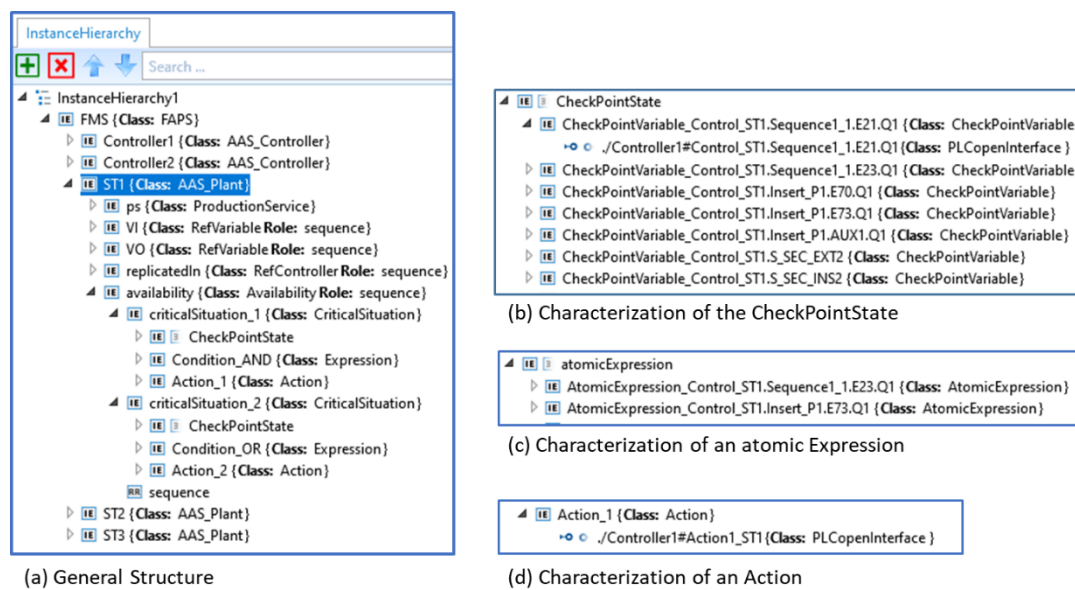


Figure 8. Flexible Automation control system design example with AutomationML Editor: (a) General Structure composed by three stations (ST1 ... ST3) controlled by two Controllers; (b) Characterization of the CheckPointState of the first critical situation identified in ST1; (c) Characterization of an atomic Expression: $\text{Control_ST1.Sequence1_1.E23.Q1} = 1$ AND $\text{Control_ST1.Insert_P1.E73.Q1} = 1$; and (d) Characterization of an Action.

3.2. Flexible Automation Projects Code Generator

The code generation of Flexible Automation Project covers normal operation as well as reconfiguration needs. As current IEC 61131-3 standard execution environments do not support dynamic code deployment, the reconfiguration requires the de-activation of a Production Service in a controller and its activation in another one. Therefore, the generated automation projects not only contain all the Production Service POU's that the controller can run, but they are enhanced with a wrapper that allows the APlant_AASs to activate/deactivate their execution. In addition, these projects also include the code to read/write the state of the Production Service of each APlant_AAS.

The APlant_AAS interacts with its associated Production Service through a predefined area of the controller memory. To that end, specific libraries are required depending on the manufacturer (e.g., S7-300 controllers from the German manufacturer Siemens require from libnodave and s7netplus libraries to enable an external access [61,62], whereas the Automation Device Specification, or ADS, is required in Beckhoff controllers for that purpose [63]).

To sum up, the Flexible Automation Projects include both the code endorsing the Production Services and the control code that supports their flexibility (see ProductionService submodel in Figure 2). Hence, each Production Service (PS) module is composed of three different POU's:

- *ProductionService_id*: a program to control the execution of the Production Service (PS_id);
- *SendStateService_id*: a program that reads and serializes the state variables of the production service;
- *ReceiveStateService_id*: a program that de-serializes the received information and updates de state variables of the production service with new initialization values in case it changes from tracking to active in a controller.

3.2.1. Production Service Program

Thanks to this program, the APlant_AAS can manage its corresponding PS, as it provides the external access required to activate/deactivate the execution of the logic and recovery/stop actions.

The program structure and the templates to be fulfilled by the generator are depicted in Figure 9. The program interface is a set of application dependent variables and other local static variables that allow APlant_AAS to manage it. These local variables are:

- *isActive* and *wasActive*: these boolean variables determine the activation/deactivation of the logic.
- Two further local variables are included to support the availability:
- *recoveryAction*: it is related to the coded actions required to manage a concrete critical situation if necessary.
- *Action_CriticalSituationID*: it identifies a specific recovery code (POU instance).

Section	Description						
Interface	<pre> interface localVars retain true variable (3) name type initialValue 1 isActive type initialValue 2 wasActive type initialValue 3 recoveryAction type initialValue </pre> <div style="display: flex; align-items: center; justify-content: center;"> <div style="font-size: 2em; margin-right: 10px;">+</div> <div> <p>Actions</p> <p>(POU instance variables)</p> </div> </div>						
Body	<p>General structure</p> <pre> IF isActive=TRUE and wasActive=TRUE THEN PS_id(); SendStateService_id (); ELSE IF isActive=TRUE and wasActive=FALSE THEN CASE recoveryAction OF /* the list value depends on the Production Service*/ /* the list cases depend on the critical situations of the Production Service */ END_CASE ELSE IF isActive=FALSE and wasActive=TRUE THEN wasActive=FALSE; END_IF </pre>						
	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="width: 33%;">Direct recovery</th> <th style="width: 33%;">Check Point Recovery</th> <th style="width: 33%;">Safe Stop</th> </tr> </thead> <tbody> <tr> <td> <pre> ReceiveStateService_id(); PS_id (); SendStateService_id (); wasActive=TRUE; </pre> </td> <td> <pre> PS_id_Action_id(); IF(PS_id_Action_id.en d) THEN ReceiveStateService_i d); PS_id(); </pre> </td> <td> <pre> PSid_Action_id(); IF(PSid_Action_id.en d) THEN recoveryAction=0; isActive=FALSE; END_IF </pre> </td> </tr> </tbody> </table>	Direct recovery	Check Point Recovery	Safe Stop	<pre> ReceiveStateService_id(); PS_id (); SendStateService_id (); wasActive=TRUE; </pre>	<pre> PS_id_Action_id(); IF(PS_id_Action_id.en d) THEN ReceiveStateService_i d); PS_id(); </pre>	<pre> PSid_Action_id(); IF(PSid_Action_id.en d) THEN recoveryAction=0; isActive=FALSE; END_IF </pre>
	Direct recovery	Check Point Recovery	Safe Stop				
<pre> ReceiveStateService_id(); PS_id (); SendStateService_id (); wasActive=TRUE; </pre>	<pre> PS_id_Action_id(); IF(PS_id_Action_id.en d) THEN ReceiveStateService_i d); PS_id(); </pre>	<pre> PSid_Action_id(); IF(PSid_Action_id.en d) THEN recoveryAction=0; isActive=FALSE; END_IF </pre>					
<pre> ReceiveStateService_id(); PS_id (); SendStateService_id (); wasActive=TRUE; </pre>							

Figure 9. General structure of a ProductionService_id program: the interface collects the parameters to configure the program. The body illustrates the skeleton of the program in ST programming language of the IEC 61131-3 standard and different sequences of actions to perform depending on the type of critical situation.

This program is automatically generated from FAPS model by M2M transformation rules. Three transformation rules have been developed: one for generating the Interface of such POU; other for its functionality (Body) and the third for the recovery actions:

- Rule 1—Interface definition: It is applied to every *InternalElement* having *RefBaseSystemUnitPath* property with *AAS_Plant* value. The rule starts adding the common part with the fixed local variables at their initial values. After, it adds as many POU instance variables as actions defined in the *CriticalSituation* elements. For this, it searches those inherited *InternalElements* that have *RefBaseSystemUnitPath* property with *Action*, getting the value of its *ExternalInterface*. This will be the type of the new added variable. The name will be the same as the *InternalElement*'s name.

- Rule 2—Body: It is applied to every *InternalElement* having *RefBaseSystemUnitPath* property with *AAS_Plant* value. The common minimal structure is initially added (See Figure 9), changing *PS_id()* by the name of the *ExternalElement* in a *POUInstance*. Furthermore, this template applies Rule 3 to complete the list of the possible causes related to the activation of a *Production Service*.
- Rule 3—Recovery Actions: It is applied to *InternalElements* that have *RefBaseSystemUnitPath* property with *CriticalSituation* value. The code to add depends on the value of the *recoveryType* property (see Figure 9). The *PS_id()* is customized following the procedure commented above.

Figure 10 exemplifies the generation process for the control program corresponding of a *Production Service* (CL1_ST_Control). The left side of the figure presents the flexible model through which the developer defines the flexible manufacturing system, while the right one shows the resultant program, *ProductionService_id*, in *PLCopen XML* format.

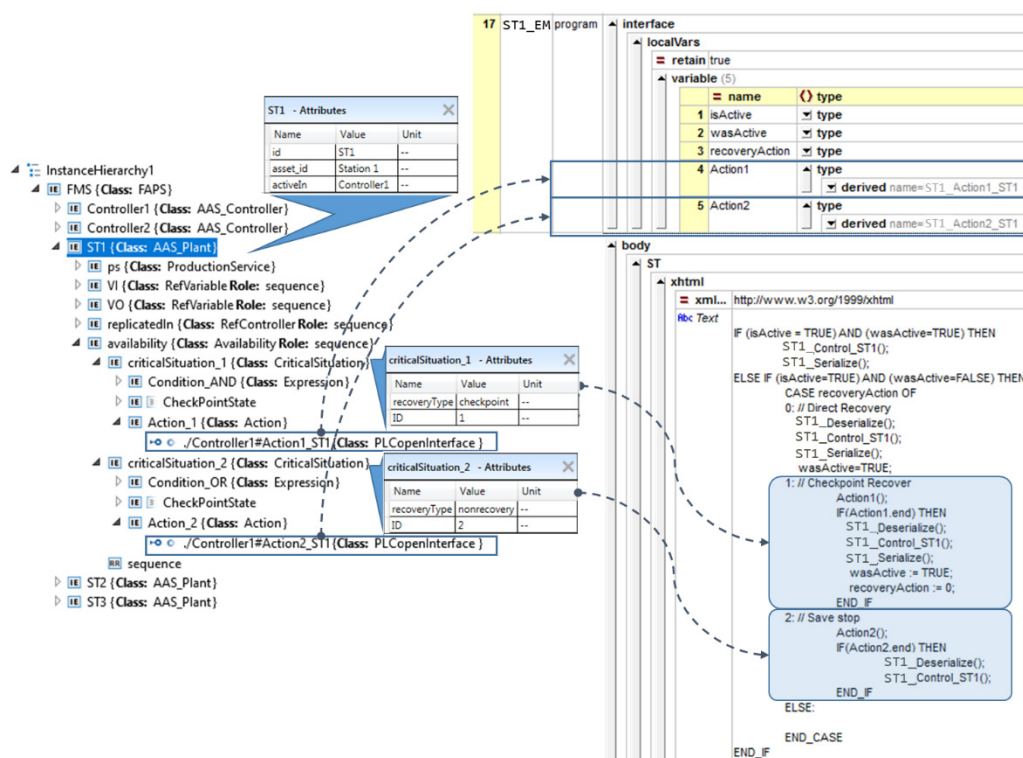


Figure 10. Example of the execution control program generation: the upper-right part of the figure shows the interface of the program. The lower-right part of the figure shows the body of the program in ST programming language of the IEC 61131-3 standard. The statements 1 and 2 of the case structure use the designated code sections for the *recoveryType* of actions 1 and 2.

3.2.2. Serialization Programs

The serialization program (*SendStateService_id*) collects the values of the state variables into a byte array, which is accessible by the *APlant_AAS*. Byte array is selected due to most IEC 61131-3 environments endorse transformation functions to cast any data type to byte (e.g., *INT_TO_BYTE*, *BOOL_TO_BYTE*, etc.).

On the contrary, the de-serialization program (*ReceiveStateService_id*) process the array sent by the *APlant_AAS* and updates the state of the *production service*.

The structure and the templates to be fulfilled by the generator are depicted in order to generate *SendStateService_id* and *ReceiveStateService_id* programs are depicted in Figure 11.

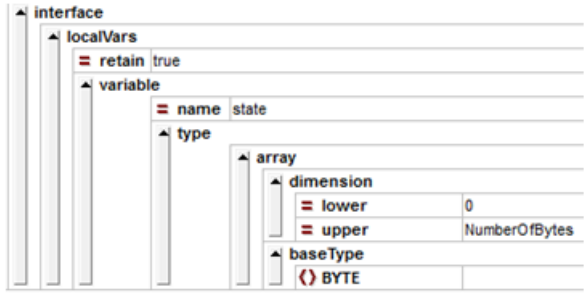
Section	Description
Interface	
Body	Serialize state[0]:=TypeOfGlobalVariable_TO_BYTE(GlobalVariable); ... state[NumberOfBytes]:=TypeOfGlobalVariable_TO_BYTE(GlobalVariable);
	De-serialize GlobalVariable=BYTE_TO_TypeOfGlobalVariable(state[0]); ... GlobalVariable = BYTE_TO_TypeOfGlobalVariable (state[NumberOfBytes]);

Figure 11. General structure of SendStateService_id and ReceiveStateService_id programs: the interface box shows the parameters to configure the program (in this case, the number of bytes of the array). The body box presents the code of the serialization and deserialization programs in ST programming language of the IEC 61131-3 standard.

These programs are automatically generated from FAPS model by M2M transformation rules. Three transformation rules have been developed: one for generating the Interface of such POU; other for serialization functionality (Body) and the third for the de-serialization body:

- Rule1—Interface Definition: It is applied to every *InternalElement* having *RefBaseSystemUnitPath* property with *POUInstance*. It initially calculates the number of bytes needed for defining the state (NumberOfBytes of Figure 11). For this, local and global variables as well as input and output parameters of the POU that implements the control logic of Production Service are identified. This POU is located in the name of the ExternalElement. Then, Rule 2 and Rule 3 are applied in order to generate the body of serialize or deserialize, respectively.
- Rule 2—Serialize Body. It requires the Production Service’s state and its corresponding variables (see Figure 11).
- Rule 3—De-Serialize Body: It also requires the state and the related variables, resulting in the writing of the new state (see Figure 11).

An example of FAP generation containing the POU, data types, global variables and tasks associated to three Production Services (ST1–ST3) is presented in Figure 12.

3.3. Application Dependant Agents Code Generator

In order to make FAM generic and customizable, this paper proposes templates for the application agents, that can be customized for specific processes. The following subsections detail such templates and the automatic generation of APlant_AASs and AController_AASs.

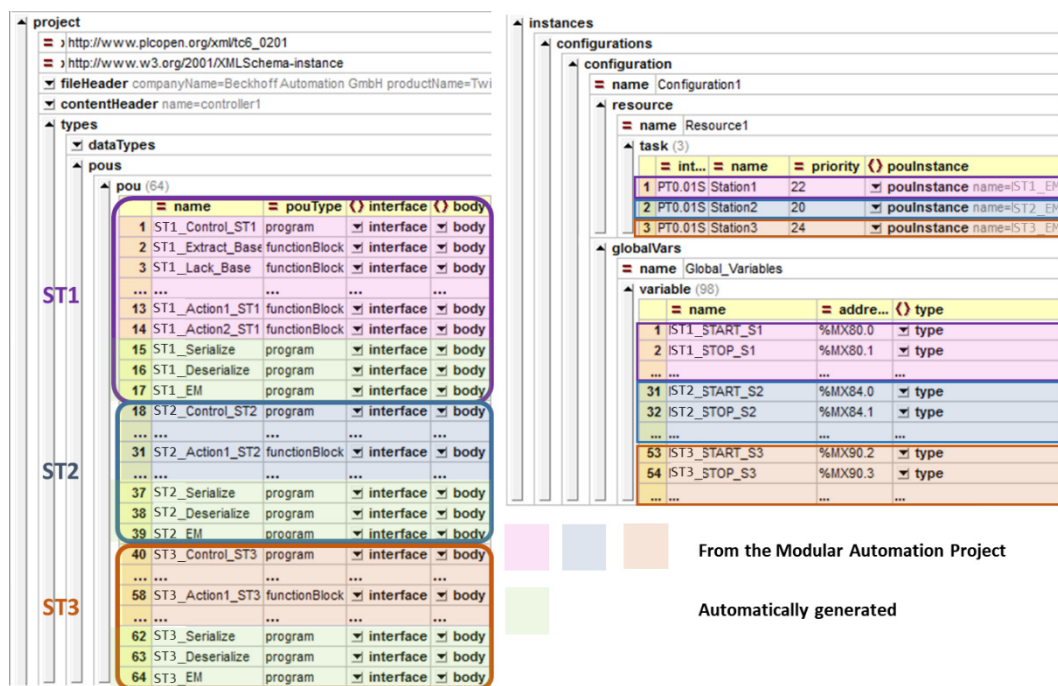


Figure 12. Example of a Flexible Automation Project containing the POUs, data types, global variables and tasks associated to three Production Services (ST1–ST3).

3.3.1. APlant_AAS Template

Each APlant_AAS is associated to several Production Services hosted in different PLCs and it performs state diagnosis, when required, for determining if the current state indicates a critical situation. The APlant_AAS template (AAS_PTemplate) proposed by the authors, presented in Figure 13, addresses this issue offering a generic and customizable solution.

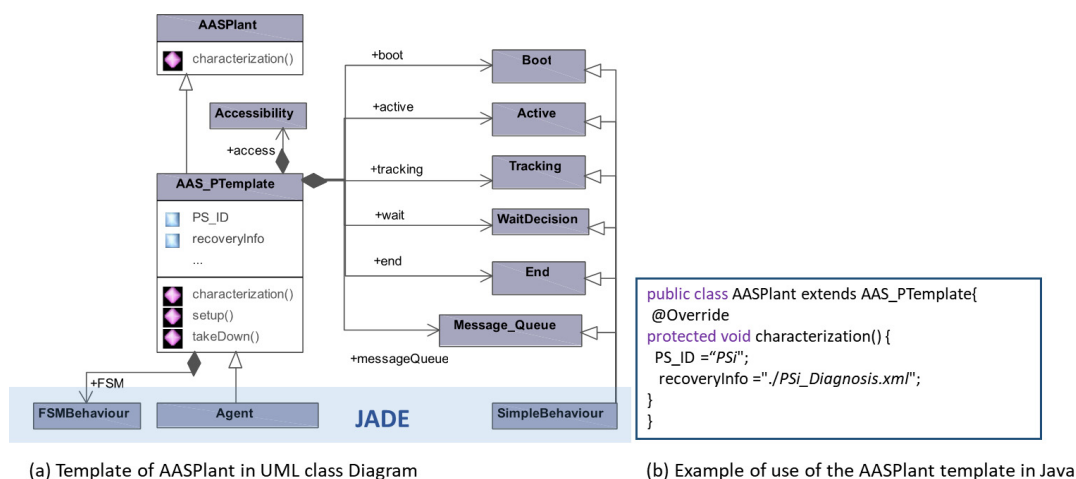


Figure 13. (a) Template of an AASPlant (AAS_PTemplate) in UML Class Diagram with detail of its methods and the states defined in its FSM and (b) Example of use of the AAS_PTemplate in Java. The characterization method of the AASPlant element allows the customization of the parameters PS_ID and recoveryInfo for different AASPlant instances.

The template can be customized by means of two parameters:

- PS_ID: this parameter contains an identifier that can match the identifier of the automation project, and that identifies each Program Organization Unit.
- recoveryInfo: it contains the set of masks to be applied to each component to determine which type of operation must be applied at any moment. These masks, defined

following the meta-model presented in Figure 14, allow to identify the critical situations of the Production System (e.g., the manufacturing steps 42–44 from Figure 3 are identified as critical situations based on the information in the recoveryInfo parameter). The Diagnosis XML file has been conceived to ease the generation and storage of such information with a predefined structure. This file stores information about the state variables (name and type), and the masks to perform both the diagnosis and the checkpoint. Note, that the diagnosis masks determine if it is a critical state type (checkpoint or unrecoverable) by filtering the state variables related to the condition to be diagnosed.

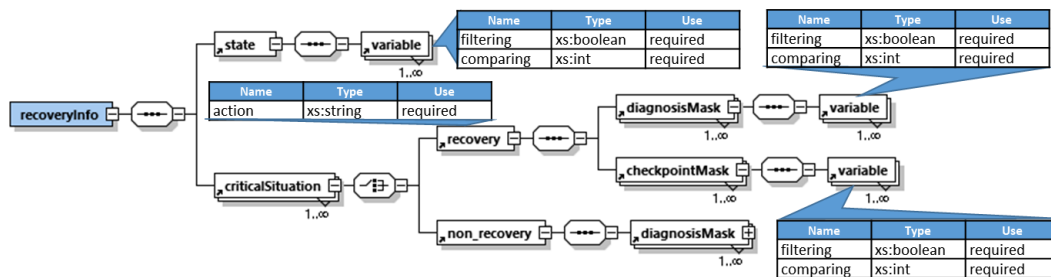


Figure 14. General Structure of the Diagnosis.xml file: Recovery information comprises the set of state variables to be analyzed as well as the set of critical situation masks. These masks allow to diagnose either recovery or non-recovery situations under controller failures/workloads balancing. This is performed by filtering a concrete set of state variables related to the condition to be diagnosed. This graphic was generated using Altova XMLSpy Version 2020.sp1.

The Component Manager of APlant_AAS has been implemented in a FSM as established in [26], which consists of the Boot, Active, Tracking, Wait decision and End states. This FSM is implemented as a JADE FSM. There are two JADE behaviors (Simple Behaviour) associated to each FSM state: one that manages the message exchange with the middleware agents (*Message_Queue*), and another one to implement the specific functionality of each state (*Boot*, *Active*, *Tracking*, *WaitDecision* and *End*). Meanwhile, *access* provides access to the PS's code in the PLC.

3.3.2. APlant_AAS Generation

The transformation of critical situations related information into a set of masks to diagnose the Production Service is a major issue in APlant_AAS generation.

To generate *PSid_Diagnosis.xml* file from FAPS model the following transformation rules are required:

- Rule 1—State characterization: It generates the set of variables conforming the state. To do this, every *InternalElement* having *RefBaseSystemUnitPath* property with *RefVariable* and the *InternalElement* having *RefBaseSystemUnitPath* property with *POUinstance* in *AAS_Plant* are processed.
- Rule 2—Critical Situation: It applies to the *InternalElements* that have *RefBaseSystemUnitPath* property with *CriticalSituation* in an *AAS_Plant*. As a result, the Critical Situation information stored in diagnosis XML file is generated.
- Rule 3—Diagnosis: It processes the Condition to identify which the state variables are required to determine the type of critical state at that situation. This rule applies to every *InternalElement* with *RefBaseSystemUnitPath* property having an Expression in a *CriticalSituation*.
- Rule 4—Checkpoint: It processes the *CheckpointState* to determine at which condition the *AAS_Plant* must be restarted. This rule applies to each *InternalElement* that have *RefBaseSystemUnitPath* property with *CheckPointVariable* in a Critical Interval.

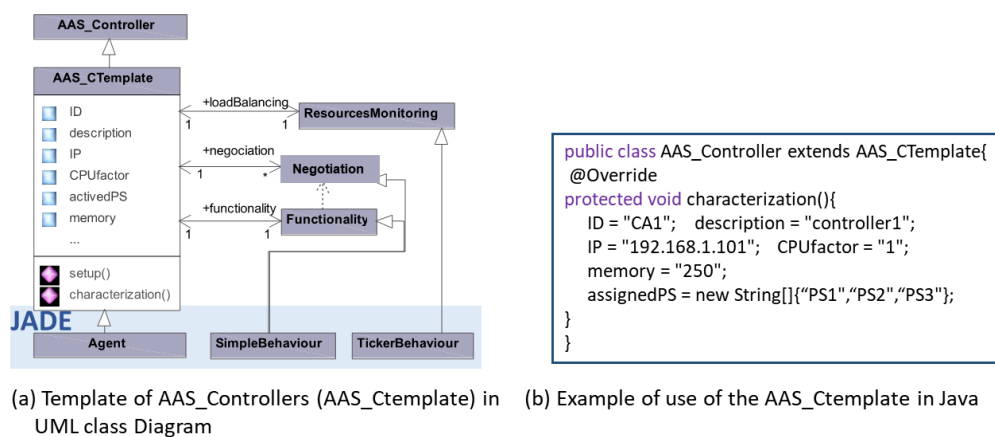
3.3.3. AController_AAS Template

The AController_AAS updates the information related to the state of the controller resources required by QoS Monitor agents. They can participate in negotiation processes

when needed. Negotiation criterion depends on the specific QoS. For instance, in Availability and after a controller failure, the D&D agent will require a negotiation process among controllers able to run the affected Production Services, being the specific criterion, for example, the minimum execution cycle.

In order to offer a generic and customizable solution, the AController_AAS template (AAS_CTemplate) illustrated in Figure 15 is proposed. The template has a set of customizable parameters:

- ID, which identifies the agent in the system;
- A textual Description;
- CPUfactor, with respect to a reference controller;
- Memory resources;
- IP address;
- AssignedPS: a list of Production Services, whose control logic is executed in the controller.



(a) Template of AAS_Controllers (AAS_Ctemplate) in UML class Diagram (b) Example of use of the AAS_Ctemplate in Java

Figure 15. (a) Template of an AAS_Controller (AAS_CTemplate) in UML Class Diagram with detail of its methods and the states defined in its FSM and (b) Example of the use of the AAS_CTemplate in Java. The characterization method of the AAS_Controller element allows the customization of the parameters ID, description, IP, CPUfactor, memory and assignedPS for different AAS_Controller instances.

The Class Diagram presented in Figure 15 defines the template for the design and parameterization of every AController_AAS of the system. The basic functionality of the AController_AAS, which manages the messages from the middleware agents, is implemented in a cyclic behavior (*functionality*). These messages can be either negotiation messages or queries about the controller resources. Each time a negotiation message is received, a new negotiation behavior is instantiated. This behavior is deleted once the negotiation process, it was related to, is concluded. Furthermore, the AController_AAS can also implement resource monitoring behaviors. These behaviors allow monitoring a specific resource of the controller as part of the QoS monitoring process.

The registration of the AController_AAS and the creation of resource monitoring and functionality behaviors are performed during the setup method of the AController_AAS.

4. Assessment

The modeling approach and the application agents presented in the previous section have been implemented in a demonstrator located at the Department of Automatic Control and Systems Engineering of the University of the Basque Country, Bilbao, Spain. This case study seeks to prove the capability to assure work balance among the distributed controllers comprising the automation system for the flexible manufacturing cell FMS-200.

The manufacturing cell comprises four stations, connected by a conveyor system, that assemble a product from a set of four parts: base, bearing, shaft and lid. The first station validates the orientation of the base, which is provided from a buffer. If the position is wrong, the base is discarded, and a new one is provided. Otherwise, the base is transferred

to the conveyor system. In the second one, a robotic arm inserts the bearing and shaft in the base, whereas the lid is placed in the third station. The fourth station acts as a warehouse, where the assembled products fed by the conveyor system are stored.

The cell is organized in five *PlantAssets*, corresponding to the four stations and the transfer system, respectively. Nevertheless, for simplicity, this assessment only considers the *PlantAssets* associated to the first three stations. Besides the manufacturing cell, the demonstrator includes two CX1020 Soft PLCs from the German manufacturer Beckhoff. These devices are characterized for their ability to run a Windows Embedded CE operating system in parallel with the Beckhoff PLC runtime. The demonstrator also contains a supervisor PC hosting the Middleware Manager and the QoS Manager.

In the first station, once the orientation of the base has been checked, a pneumatic suction gripper is responsible of picking the base and placing it in the conveyor system. Any interruption during the execution of this task implies a loss of reference of the current state, and therefore, it is considered a critical operation. The reconfiguration actions to be considered will differ depending on the position of the gripper when the incident occurs. Thus, two different critical situations have been defined. In case the base falls during the initial lifting (i.e., before the gripper starts moving towards the conveyor system), it is retired from the station and a new base is supplied. On the other hand, if the gripper is already moving towards the conveyor system when the failure arises, the system cannot assure the return to a previous known state by itself. Thus, the system goes to a safe stop state, triggering an alarm to warn the operator of the problem.

The second station uses a robot arm to place the bearing and shaft on the base. The communication with the robot arm generates six critical situations, in which the PLC that holds the corresponding production service, does not know the position of the robot. These situations are defined as checkpoint situations in which the connection to the robot needs to be recovered and the execution of the code resumed.

The third station completes the assembly of the product by placing the lid. The composition of the lids can vary in terms of color, material, and height. Hence, this station has different actuators to introduce lids in the station, to place them in the product, or to remove them in case they do not match the product to be assembled. These actuators can perform their operations in parallel, as they are allocated around a rotary table, and they are considered critical operations. Up to five critical situations have been identified, with their subsequent actions to resume a normal execution. It must be noted that each critical situation must consider the execution states of all the actuators performing parallel operations.

The minimal POU's of the control code, generated following the methodology presented in [48], are stored in a model-oriented database (see Figure 6). On the other hand, developers with AML-based editor design Flexible Automation Production System as a set of Production Services. They also specify the identified critical situations. Part of the complete model is presented in Figure 8.

The application of the transformation rules generates the flexible automation project of each PLC in the system containing the POU's, data types and global variables related to the production services it can offer. The code includes the POU that manages the activation/deactivation of the Control Logic (CL), as well as state serialization FBs. The different parts of the Production Services contained in the flexible automation project for controller 1 are presented in Figures 10 and 12.

Concerning the application agents, the Flexible Automation Framework generates the corresponding AController_AASs (see example in Figure 15), APlant_AASs (see example in Figure 13) and their corresponding diagnosis files.

As a result of the automatic code generation, the Flexible Automation Projects, as well as the AController_AAs, APlant_AASs and the diagnosis information files, are deployed into the controllers.

The assessment process comprises the analysis of two different features: evaluation of the reconfiguration capabilities using a concrete example, and the scalability of the proposed approach.

The assessment of reconfiguration capabilities consisted of the following: initially, there was a unique control system (controller 1) in charge of the execution of the automation control software of the three APlant_Assets. At a certain point, a second controller (controller 2) joined the system.

Figure 16 illustrates the sequence of interactions between agents in the assessed scenario. The reconfiguration process is divided in three steps:

- Step 1—QoS Loss Detection: Registration of controller 2 unbalances the system as its current workload is too low. When the workload monitoring of controller 2 detects it, the AController_AAS2 (CA2 in Figure 16) triggers an event to notify the QM (“QoS_Loss_Event(lowerLimit_reach;CA2)” in Figure 16), that eventually leads to Production System reconfiguration (“QoS_Reconfiguration_Event(systemLoad;lowerLimit_recovery;CA2)” in Figure 16).
- Step 2—Diagnosis and Decision: The D&D receives the reconfiguration event and proceeds to initiate a negotiation among controllers to decide the new distribution. For simplicity, this negotiation process is encapsulated in the “Workload Optimization Process” block in Figure 16. The new distribution depends on the CPU factors of the AController_AASs (CA1 and CA2 in Figure 16), current distribution of the Production Services, and the CPU workload limits introduced by the Production Service software modules. When the negotiation concludes, the D&D launches the relocation of the Production Services).
- Step 3—Reconfiguration: Relocation is represented by the “for loop: PSs to be relocated” in Figure 16, which consists of the following steps. Firstly, the D&D forces the APlant_AASs (PSAi,j in Figure 16) of the affected production service software modules to move to wait decision state (wait in Figure 13). Similarly, the D&D also forces the former active APlant_AAS to stop in the next non-critical situation of the control code (“change_State(waitDecision;nonCriticalStop)” in Figure 16). At this point, the D&D commands the execution of the production service software modules from the last known state in their new locations (“change_State(active;direct)” in Figure 16).

After the reconfiguration process, the control software of station 1 and station 2 *PlantAssets* are running in controller 1 and the control software of station 3 is running in controller 2. Figure 17 presents a graphic depicting the workload of the different controllers before and after the introduction of controller 2.

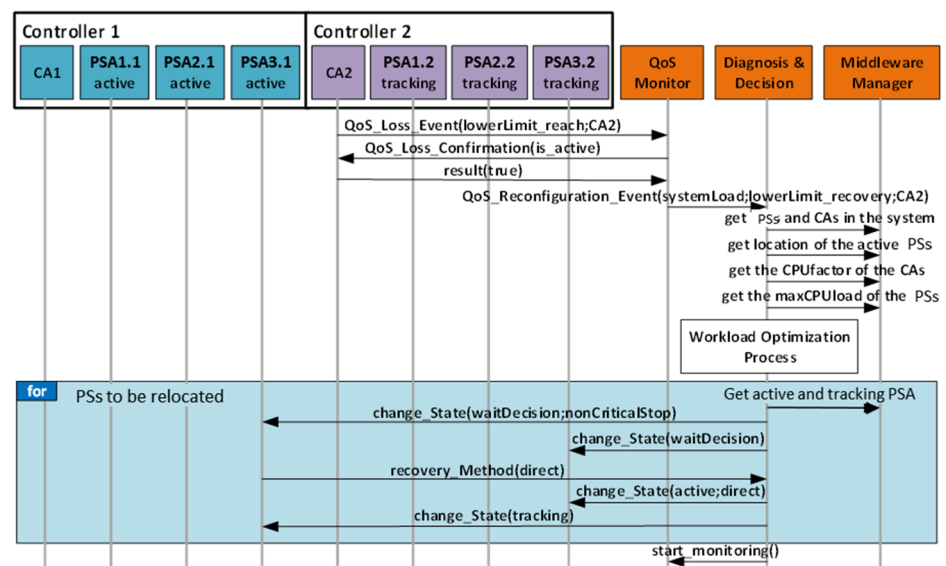


Figure 16. Sequence diagram detailing the triggering of the Workload Optimization process and the redistribution of active PSs among the controllers: when Controller 2 joins the system, CA2 informs the QoS monitor that its workload is too low, and after confirmation, the QoS Monitor sends a reconfiguration event to the D&D (Step 1). The D&D requests information to the MM and after receiving it, it triggers the Workload Optimization Process (Step 2). As a result of this optimization, the D&D stops PSA3.1 at the first non-critical situation, changes the state of PSA3.1 and PSA3.2 to “wait”, and later changes the state of PSA3.2 to “active” and the state of PSA3.1 to “tracking” (Step 3).

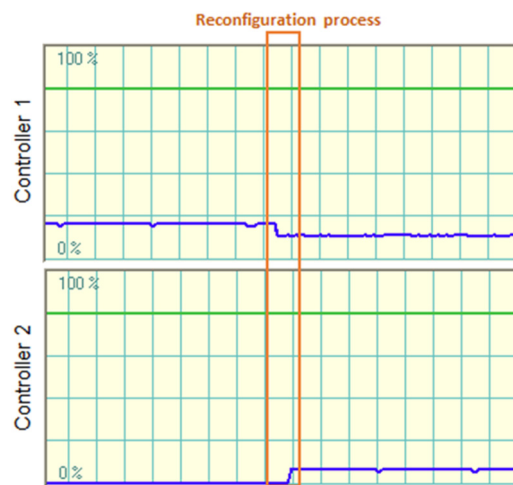


Figure 17. Workload of controller 1 and controller 2 before and after the reconfiguration: the orange box (reconfiguration process) corresponds to the for loop shadowed in blue in Figure 16. As a result of the reconfiguration process, the workload of Controller 1 decreases (now it has two active Production Services), and the workload of Controller 2 increases (it has one active Production Service).

The scalability assessment evaluates the time the D&D takes to decide the new distribution (Step 2) and the time needed by the architecture to reconfigure the system (Step 3). This depends on the number of automation software modules to be reconfigured as well as on the number of PLCs that may run the involved software modules. For this test, all need to be relocated and the reconfiguration is launched when the control software modules operate in a non-critical state. Table 1 presents the results of this test. The first column represents the time in milliseconds the D&D takes to calculate the new distribution of the *PlantAsset* software module; while the following columns present the time it takes to reconfigure each.

Table 1. Distribution algorithm and reconfiguration times.

PSAs	Dist. Alg.	1st Reconf	2nd Reconf	3rd Reconf	4th Reconf	5th Reconf	6th Reconf	7th Reconf	8th Reconf
2	5.29	610.04	1214.79						
3	6.14	610.66	1216.09	1822.70					
4	8.36	612.20	1221.51	1830.35	2437.00				
5	11.67	616.58	1236.59	1843.65	2450.43	3056.97			
6	12.72	617.41	1222.53	1827.60	2432.38	3065.81	3687.38		
7	16.98	625.51	1230.31	1837.61	2444.37	3050.74	3657.21	4262.64	
8	17.49	622.99	1242.58	1851.25	2456.50	3062.99	3667.54	4275.75	4880.56

The following figures illustrate how the execution time of the re-distribution algorithm (Figure 18) and the overall reconfiguration time (Figure 19) increase with the number of software modules to be reconfigured. However, the time to make a decision is negligible in comparison with the overall reconfiguration time, as it is computed within the D&D and it does not include negotiation. It is also remarkable that the time for reconfiguring each software module is approximately the same, around 615 ms.

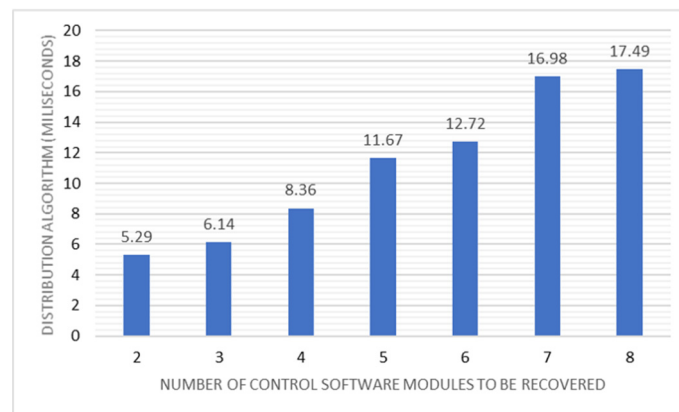


Figure 18. Distribution Algorithm Time vs. Number of control software modules.

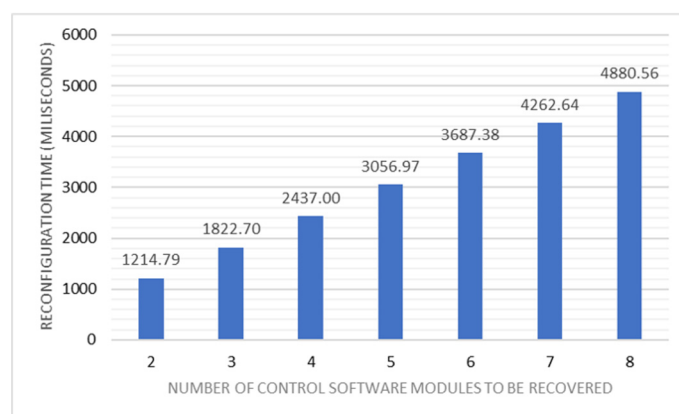


Figure 19. Reconfiguration Time vs. Number of control software modules.

5. Conclusions

This paper presents an approach aiming at adding flexibility to automation production systems following Industry 4.0 issues. It adds this flexibility by reconfiguring the control

system, i.e., relocating the different functionalities over the distributed control system, assuring the execution despite controller failure.

The proposed approach defines FAPS as a set of Controller and Plant I4.0 interconnected components, which support reconfiguration according to QoS parameters. It assumes that a MAS-based middleware provides QoS management at runtime but it offers model-based support to specify flexibility needs of target systems and automatically generate: (1) the flexible automation project for each controller in the system, as well as; (2) the code of application dependent agents, being the AAS's component manager implementation for the system's components. Furthermore, a template for application dependent agents (controller and plant) has been defined to make FAM generic and customizable. These templates can be customized for specific processes.

The core of the framework proposed by the authors is based on MDE that allows managing complex systems. In fact, a Model Editor guides designers along the design of automation production systems, offering means for characterizing the critical situations of the production services, and collecting this information in a Model. The automatic generation has been performed via M2M and M2T transformation rules having as input the model generated by the Editor. This avoids manual programming errors, very common in these such complex situations.

The execution of the assessment has allowed to put in practice the proposed approach. As a result, several conclusions have been obtained:

- The definition of the critical intervals in the manufacturing process, which is essential to manage flexibility properly, has proved to be a difficult task. To that end, it is necessary to rely on someone with a great knowledge of the manufacturing process, who has also taken part in the design process of the code for the controllers. As far as authors know, other approaches do not contemplate this task, as they consider that the state represents the whole process, and thus any situation should be recoverable. Nevertheless, the reality is very different.
- The advantages of AutomationML for modeling and processing different types of data have been demonstrated. Interoperability is ensured with AutomationML as long as the integrators use PLCOpen, which is widely known and used by an increasing number of engineers and suppliers (e.g., Siemens allows to export hardware configuration to AutomationML). Despite it is a relatively new standard, it is receiving an increasing attention as exportation format for different types of data (information, code, etc.).
- The proposed approach eases the reconfiguration and scalability of the system, allowing the reconfiguration of the control system (by adding or removing controllers) without changing the configuration of the assets. In the same way, changes in the control of the process can be easily implemented thanks to automated code generation.

However, in case the target PLC is not PLCOpen compliant, the proposed approach could not be applied. That supposes a limitation in terms of scalability of the solution at design time. Regarding future work, the use of accurate simulation models as a resource to identify critical situations of the manufacturing process in a safe and controlled environment, could allow to perform cost-opportunity analyses in order to decide whether additional sensors should be included.

Author Contributions: Conceptualization, software, validation, writing—original draft preparation, U.G.; software, validation, writing—original draft preparation, A.L.; methodology, validation, writing—review and editing, A.A.; methodology, software, writing—review and editing, E.E.; conceptualization, supervision and writing—review and editing, M.M. All authors have read and agreed to the published version of the manuscript.

Funding: This work was financed by MCIU/AEI/FEDER, UE (grant number RTI2018-096116-B-I00) and by GV/EJ (grant number IT1324-19).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The functionality of the flexible assembly cell FMS-200, located in the Department of Automatic Control and System Engineering of the University of the Basque Country is online available: <https://youtu.be/7Ifp5jD3-4U> (accessed on 11 January 2021). The availability of the Control System is online available: https://youtu.be/uK1w5p3_RQ (accessed on 11 January 2021).

Acknowledgments: The authors would like to express gratitude to the Government of Spain for its support to the research project in which this work is framed (grant number RTI2018-096116-B-I00), as well as to the Government of the Basque Country Region (grant number IT1324-19). We would also like to thank to the referees which provided us their feedback for the improvement of this manuscript. A very special thank to Rafael Priego and Birgit Vogel for their collaboration in the work.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following table collects the acronyms and abbreviations used throughout the paper.

AAS	Asset Administration Shell
AML	Automation ML
APS	Automation Production Systems
CAEX	Computer Aided Engineering eXchange
CL	Control Logic
D&D	Diagnosis & Decision
DT	Digital Twin
FAM	Flexible Automation Middleware
FAPS	Flexible Automation Production System
FAVA	Functional Application Design for Distributed Automation Systems
FB	Function Blocks
FSM	Finite State Machine
IE	Internal Element
JADE	Java Agent Development Framework
MM	Middleware Manager
MDD	Model Driven Design
MDE	Model Driven Engineering
M2M	Model to Model
M2T	Model to Text
MAS	Multi Agent Systems
PS	Production Service
PLCs	Programmable Logic Controllers
POU	Program Organization Unit
QoS	Quality of service
QM	QoS Monitor
RAMI 4.0	Reference Architecture Model for Industry 4.0
SysML	System Modeling Language
SR	System Repository
SUC	System Unit Classes
UML	Unified Modeling Language
XML	eXtensible Markup Language

References

1. European Commission. Research and Innovation. Factories of the Future PPP: Towards Competitive EU Manufacturing. Available online: https://ec.europa.eu/research/press/2013/pdf/ppp/fof_factsheet.pdf (accessed on 1 February 2021).
2. Blanchet, M.; Rinn, T.; Von Thaden, G.; de Thieulloy, G. Industry 4.0 The New Industrial Revolution How Europe Will Succeed. Available online: http://www.iberglobal.com/files/Roland_Berger_Industry.pdf (accessed on 1 February 2021).
3. National Science and Technology Council. ADVANCED MANUFACTURING: A Snapshot of Priority Technology Areas Across the Federal Government. Available online: https://www.mrs.org/docs/default-source/advocacy-policy/resources/advanced-manufacturing---A-snapshot-of-priority-technology-areas.pdf?sfvrsn=fb15e811_6 (accessed on 1 February 2021).
4. Liao, Y.; Deschamps, F.; Loures, E.F.R.; Ramos, L.F.P. Past, present and future of Industry 4.0—A systematic literature review and research agenda proposal. *Int. J. Prod. Res.* **2017**, *55*, 3609–3629. [CrossRef]

5. European Commission; European Factories of the Future Research Association (EFFRA). Factories of the Future. Multi-Annual Roadmap for the Contractual PPP under Horizon 2020. Available online: https://www.effra.eu/sites/default/files/factories_of_the_future_2020_roadmap.pdf (accessed on 1 February 2021).
6. Lindstrom, J.; Kyosti, P.; Birk, W.; Lejon, E. An initial model for zero defect manufacturing. *Appl. Sci.* **2020**, *10*, 4570. [CrossRef]
7. Mourtzis, D. Simulation in the design and operation of manufacturing systems: State of the art and new trends. *Int. J. Prod. Res.* **2020**, *58*, 1927–1949. [CrossRef]
8. Mourtzis, D.; Vlachou, E. A cloud-based cyber-physical system for adaptive shop-floor scheduling and condition-based maintenance. *J. Manuf. Syst.* **2018**, *47*, 179–198. [CrossRef]
9. Lu, Y.; Xu, X.; Wang, L. Smart manufacturing process and system automation—A critical review of the standards and envisioned scenarios. *J. Manuf. Syst.* **2020**, *56*, 312–325. [CrossRef]
10. Cotrino, A.; Sebastián, M.A.; González-Gaya, C. Industry 4.0 roadmap: Implementation for small and medium-sized enterprises. *Appl. Sci.* **2020**, *10*, 8566. [CrossRef]
11. Tay, S.I.; Malaysia, T.H.O.; Raja, P.; Pahat, B.; Hamid, N.A.A.; Ahmad, A.N.A. An overview of industry 4.0: Definition, components, and government initiatives. *J. Adv. Res. Dyn. Control. Syst.* **2018**, *10*, 1379–1387.
12. Florescu, A.; Barabas, S.A. Modeling and simulation of a flexible manufacturing system—A basic component of industry 4.0. *Appl. Sci.* **2020**, *10*, 8300. [CrossRef]
13. Shen, W.; Wang, L.; Hao, Q. Agent-based distributed manufacturing process planning and scheduling: A state-of-the-art survey. *IEEE Trans. Syst. Part. C* **2006**, *36*, 563–577. [CrossRef]
14. Krupitzer, C.; Roth, F.M.; VanSyckel, S.; Schiele, G.; Becker, C. A survey on engineering approaches for self-adaptive systems. *Pervasive Mob. Comput.* **2015**, *17*, 184–206. [CrossRef]
15. Wang, L.; Adamson, G.; Holm, M.; Moore, P. A review of function blocks for process planning and control of manufacturing equipment. *J. Manuf. Syst.* **2012**, *31*, 269–279. [CrossRef]
16. Nouri, H. Development of a comprehensive model and BFO algorithm for a dynamic cellular manufacturing system. *Appl. Math. Model.* **2016**, *40*, 1514–1531. [CrossRef]
17. Urban, T.L.; Chiang, W.-C. Designing energy-efficient serial production lines: The unpaced synchronous line-balancing problem. *Eur. J. Oper. Res.* **2016**, *248*, 789–801. [CrossRef]
18. Legat, C.; Vogel-Heuser, B. A Multi-agent architecture for compensating unforeseen failures on field control level. In *Service Orientation in Holonic and Multi-Agent Manufacturing and Robotics. Studies in Computational Intelligence*; Borangiu, T., Trentesaux, D., Thomas, A., Eds.; Springer: Cham, Switzerland, 2014; Volume 544, pp. 195–208. [CrossRef]
19. Ribeiro, L.; Barata, J.; Onori, M.; Hoos, J. Industrial agents for the fast deployment of evolvable assembly systems. In *Industrial Agents*; Leitão, P., Karnouskos, S., Eds.; Morgan Kaufmann: Boston, MA, USA, 2015; pp. 301–322, ISBN 9780128003411. [CrossRef]
20. Rocha, A.; Di Orio, G.; Barata, J.; Antzoulatos, N.; Castro, E.; Scrimieri, D.; Ratchev, S. An agent based framework to support plug and produce. In Proceedings of the 2014 12th IEEE International Conference on Industrial Informatics (INDIN 2014), Porto Alegre, Brazil, 27–30 July 2014; pp. 504–510. [CrossRef]
21. Botygin, I.A.; Tartakovskiy, V.A. The development and simulation research of load balancing algorithm in network infra-structures. In Proceedings of the 2014 International Conference on Mechanical Engineering, Automation and Control Systems (MEACS 2014), Tomsk, Russia, 16–18 October 2014; pp. 1–5. [CrossRef]
22. Guo, L.; Wang, B.; Wang, W. Research of energy-efficiency algorithm based on on-demand load balancing for wireless sensor networks. In Proceedings of the 2009 International Conference on Test and Measurement, Hong Kong, China, 5–6 December 2009; pp. 22–26. [CrossRef]
23. Merz, M.; Frank, T.; Vogel-Heuser, B. Dynamic redeployment of control software in distributed industrial automation systems during runtime. In Proceedings of the 2012 IEEE International Conference on Automation Science and Engineering (CASE 2012), Seoul, Korea, 20–24 August 2012; pp. 863–868. [CrossRef]
24. Streit, A.; Rösch, S.; Vogel-Heuser, B. Redeployment of control software during runtime for modular automation systems taking real-time and distributed I/O into consideration. In Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA 2014), Barcelona, Spain, 16–19 September 2014; pp. 1–4. [CrossRef]
25. Salazar, L.A.C.; Mayer, F.; Schütz, D.; Vogel-Heuser, B. Platform independent multi-agent system for robust networks of production systems. *IFAC PapersOnLine* **2018**, *51*, 1261–1268. [CrossRef]
26. Priego, R.; Iriondo, N.; Gangoiti, U.; Marcos, M. Agent Based Middleware Architecture for Reconfigurable Manufacturing Systems. *Int. J. Adv. Manuf. Technol.* **2017**, *92*, 1579–1590. [CrossRef]
27. International Electrotechnical Commission. Smart Manufacturing—Reference Architecture Model Industry 4.0 (RAMI4.0). IEC Standard PAS 63088: 2017(E). Available online: <https://webstore.iec.ch/publication/30082> (accessed on 3 February 2021).
28. Wang, H. Dynamic Fault Handling and Reconfiguration for Industrial Automation Systems. Available online: https://www.ias.uni-stuttgart.de/dokumente/publikationen/2019_Dynamic_Fault_Handling_and_Reconfiguration_for_Industrial_Automation_Systems.pdf (accessed on 3 February 2021).
29. Lyu, G.; Fazlirad, A.; Brennan, R.W. Multi-agent modeling of cyber-physical systems for IEC 61499 based distributed automation. *Procedia Manuf.* **2020**, *51*, 1200–1206. [CrossRef]
30. Fraile, F.; Sanchis, R.; Poler, R.; Ortiz, A. Reference models for digital manufacturing platforms. *Appl. Sci.* **2019**, *9*, 4433. [CrossRef]

31. Cavaliere, S.; Salafia, M.G. Insights into mapping solutions based on OPC UA information model applied to the industry 4.0 asset administration shell. *Computers* **2020**, *9*, 28. [CrossRef]
32. Cavaliere, S.; Giuseppe, M.G. Asset administration shell for PLC representation based on IEC 61131-3. *IEEE Access* **2020**, *8*, 142606–142621. [CrossRef]
33. Glossary. Available online: https://www.plattform-i40.de/SiteGlobals/PI40/Forms/Listen/Glossar/EN/Glossary_Formular.html?queryResultId=null&pageNo=0&resourceId=1081500&pageLocale=en&input_=1081494&titlePrefix=Alle (accessed on 13 February 2021).
34. International Electrotechnical Commission. IEC 61131-3:2013 Programmable Controllers—Part 3: Programming Languages. Available online: <https://webstore.iec.ch/publication/4552> (accessed on 3 February 2021).
35. The Structure of the Administration Shell: Trilateral Perspectives from France, Italy and Germany. Available online: https://www.plattform-i40.de/PI40/Redaktion/EN/Downloads/Publikation/hm-2018-trilaterale-coop.pdf?__blob=publicationFile&v=4 (accessed on 25 February 2021).
36. Booch, G.; Rumbaugh, J.; Jacobson, I. *The Unified Modeling Language User Guide*, 2nd ed.; Addison-Wesley Professional: Boston, MA, USA, 2015; ISBN 0321267974.
37. Estevez, E.; Marcos, M.; Gangoiti, U.; Orive, D. A Tool Integration Framework for Industrial Distributed Control Systems. In Proceedings of the 44th IEEE Conference on Decision and Control, Seville, Spain, 12–15 December 2005; pp. 8373–8378. [CrossRef]
38. Hästbacka, D.; Vepsäläinen, T.; Kuikka, S. Model-driven development of industrial process control applications. *J. Syst. Softw.* **2011**, *84*, 1100–1113. [CrossRef]
39. Thramboulidis, K.; Frey, G. Towards a model-driven IEC 61131-based development process in industrial automation. *J. Softw. Eng. Appl.* **2011**, *4*, 217–226. [CrossRef]
40. Vyatkin, V.; Hanisch, H.-M.; Pang, C.; Yang, C.-H. Closed-loop modeling in future automation system engineering and validation. *IEEE Trans. Syst. Part. C* **2009**, *39*, 17–28. [CrossRef]
41. SysML. The SysML Specification. Available online: <http://www.sysml.org> (accessed on 3 February 2021).
42. Schütz, D.; Obermeier, M.; Vogel-heuser, B. SysML-based approach for automation software development—Explorative usability evaluation of the provided notation. In *Design, User Experience, and Usability. Web, Mobile, and Product Design. DUXU 2013*; Lecture Notes in Computer Science; Marcus, A., Ed.; Springer: Berlin/Heidelberg, Germany, 2013; Volume 8015, pp. 568–574. [CrossRef]
43. Fay, A.; Vogel-Heuser, B.; Frank, T.; Eckert, K.; Hadlich, T.; Diedrich, C. Enhancing a model-based engineering approach for distributed manufacturing automation systems with characteristics and design patterns. *J. Syst. Softw.* **2015**, *101*, 221–235. [CrossRef]
44. Wehrmeister, M.A.; de Freitas, E.P.; Binotto, A.P.D.; Pereira, C.E. Combining aspects and object-orientation in model-driven engineering for distributed industrial mechatronics systems. *Mechatronics* **2014**, *24*, 844–865. [CrossRef]
45. Marcos, M.; Estevez, E.; Perez, F.; Van der Wal, E. XML exchange of control programs. *IEEE Ind. Electron. Mag.* **2009**, *3*, 32–35. [CrossRef]
46. Van der Wal, E. PLCopen. *IEEE Ind. Electron. Mag.* **2009**, *3*, 25. [CrossRef]
47. Thramboulidis, K. The 3+1 SysML view-model in model integrated mechatronics. *J. Softw. Eng. Appl.* **2010**, *3*, 109–118. [CrossRef]
48. Priego, R.; Armentia, A.; Estévez, E.; Marcos, M. Modeling techniques as applied to generating tool-independent automation projects. *Automatisierungstechnik* **2016**, *64*, 325–340. [CrossRef]
49. Vogel-Heuser, B.; Schütz, D.; Frank, T.; Legat, C. Model-driven engineering of Manufacturing Automation Software Projects—A SysML-based approach. *Mechatronics* **2014**, *24*, 883–897. [CrossRef]
50. Institute of Automation and Information Systems. Functional Application Design for Distributed Automation Systems (FAVA). Available online: <https://www.ais.mw.tum.de/en/research/> (accessed on 3 February 2021).
51. Vogel-Heuser, B.; Rösch, S. Integrated modeling of complex production automation systems to increase dependability. In *Risk—A Multidisciplinary Introduction*; Klüppelberg, C., Straub, D., Welpel, I., Eds.; Springer: Cham, Switzerland, 2014; pp. 363–385. [CrossRef]
52. Cândido, G.; Colombo, A.W.; Barata, J.; Jammes, F. Service-oriented infrastructure to support the deployment of evolvable production systems. *IEEE T. Ind. Inform.* **2011**, *7*, 759–767. [CrossRef]
53. Legat, C.; Schütz, D.; Vogel-Heuser, B. Automatic generation of field control strategies for supporting (re-)engineering of manufacturing systems. *J. Intell. Manuf.* **2014**, *25*, 1101–1111. [CrossRef]
54. Selic, B. The pragmatics of model-driven development. *IEEE Softw.* **2003**, *20*, 19–25. [CrossRef]
55. Binder, C.; Neureiter, C.; Lastro, G. Towards a MDA process for developing industry 4.0 applications. *Int. J. Model. Opt.* **2019**, *9*, 1–6. [CrossRef]
56. Lüder, A.; Estévez, E.; Hundt, L.; Marcos, M. Automatic transformation of logic models within engineering of embedded mechatronic units. *Int. J. Adv. Manuf. Technol.* **2011**, *54*, 1077–1089. [CrossRef]
57. AutomationML. Available online: <http://www.automationml.org/> (accessed on 3 February 2021).
58. Schmidt, D.C. Guest editor’s introduction: Model-driven engineering. *Computer* **2006**, *39*, 25–31. [CrossRef]
59. Estévez, E.; Marcos, M. Model-based validation of industrial control systems. *IEEE Trans. Ind. Inform.* **2012**, *8*, 302–310. [CrossRef]
60. Fedai, M.; Drath, R. CAEX—A neutral data exchange format for engineering data. *ATP Int. Autom. Technol.* **2005**, *1*, 43–51.
61. Hergenbahn, T. LIBNODAVE—Exchange Data with Siemens PLCs. Available online: <http://libnodave.sourceforge.net/> (accessed on 3 February 2021).

-
62. Heiser, D.; Croes, M.; Schlameuß, R. S7netplus. Available online: <https://github.com/S7NetPlus/s7netplus> (accessed on 11 January 2021).
 63. Beckhoff. Automation Device Specification (ADS). Available online: https://infosys.beckhoff.com/english.php?content=../content/1033/tcadscommon/html/tcadscommon_intro.htm&id= (accessed on 3 February 2021).

3.3 A customizable architecture for application-centric management of context-aware applications

Gangoiti, U., López, A., Armentia, A., Estévez, E., Casquero, O., y Marcos, M. (2022). A customizable architecture for application-centric management of context-aware applications. *IEEE Access*, 10, pp. 1603-1625.

DOI: <https://doi.org/10.1109/ACCESS.2021.3138586>.

JCR©2021: 3,476

Categoría: Engineering, Electrical & Electronic

Cuartil: Q2 (105/276)

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.Doi Number

A customizable architecture for application-centric management of context-aware applications

Unai Gangoit¹, Alejandro López¹, Aintzane Armentia¹, Elisabet Estévez², Oskar Casquero¹ and Marga Marcos¹, Senior Member, IEEE

¹Systems Engineering and Automatic Control Department, University of the Basque Country (UPV/EHU), Bilbao 48013, Spain

²Electronics and Automation Engineering Department, University of Jaén, Jaén 23071, Spain

Corresponding author: A. Armentia (e-mail: aintzane.armentia@ehu.eus).

This work was financed in part by MCIU/AEI/FEDER, UE under Grant RTI2018-096116-B-I00 and in part by GV/EJ under Grant IT1324-19

ABSTRACT Context-aware applications present common requirements (e.g., heterogeneity, scalability, adaptability, availability) in a variety of domains (e.g., healthcare, natural disaster prevention, smart factories). Besides, they do also present domain specific requirements, among which the application concept itself is included. Therefore, a platform in charge of managing their execution must be generic enough to cover common requirements, but it must also be adaptable enough to consider the domain aspects to meet the demands at application-level. Several approaches in the literature tackle some of these demands, but not all of them, and without considering the applications concept and the customization demands in different domains. This work proposes a generic and customizable management architecture that covers both types of requirements based on multi-agent technology and model-driven development. Multi-agent technology is used to enable the distributed intelligence needed to address many common requirements, whereas model-driven development allows to address domain specific particularities. On top of that, a customization methodology to develop specific platforms from this generic architecture is also presented. This methodology is assessed by means of a case study in the domain of eHealthCare. Finally, the performance of MAS-RECON is compared with the most popular tool for the orchestration of containerized applications.

INDEX TERMS Application-centric management, application-driven adaptability, context-aware applications, customizable management architecture, multi-agent systems, stateful availability

I. INTRODUCTION

Current advances on information and communication technologies have allowed the expansion of the Internet of Things (IoT) [1], [2] as well as of its industrial variation, Industrial IoT (IIoT) [3], [4]. These paradigms are based on the universal interconnection of “objects” or “things” endowed with digital entities with the ability to measure or to process data, which allows the development of context-aware applications. Context-aware applications monitor their context to capture data that can be used just for supervisory purposes or for detecting abnormal situations with the aim of preventing or reacting to them. All this without human intervention. These context-aware applications belong to very different application domains, ranging from remote monitoring for natural disaster prevention [5], [6] or medical supervision [7],

[8], to smart agriculture [9], [10] or flexible manufacturing systems (FMS) [11], [12].

Such different applications have some common requirements, as illustrated in Table I. Context data are usually captured by embedded devices close to the physical environment, whereas processing tasks may require high performance equipment that is usually located far away (*distribution and node heterogeneity*). Therefore, these applications consist of different pervasive components that must communicate with each other, sometimes with time constraints (*timing requirements*). These applications might need to evolve with context changes (*adaptability*). Furthermore, sometimes co-operation among different applications is necessary to monitor the environment and/or to react to changes in it. As a result, applications and resources

may join or leave over time (*scalability*), changing resource demand and/or availability accordingly. Finally, due to the sensitive nature of the captured and processed information, apart from securing data (*security and privacy*), it is also essential to minimize service interruption and recovering application execution from the stop point, even in case of node failure (*service availability*).

Context-aware applications also present particularities of their own application domain, starting from the application concept itself. In eHealthCare (eHC) systems, an application can be understood as all the medical monitoring tasks needed to supervise the health of a person. However, an application in

FMS can be understood as tracking the manufacturing of a set of products. In both domains, it might be necessary to resolve unexpected events, such as detecting health deterioration or a malfunction of a manufacturing station. Application structure is also domain dependent, as applications are composed of a set of domain entities that collaborate to achieve the application functional goals. Additionally, applications can have non-functional requirements which apply to all application entities (e.g., in a wildfire detection system, the number of sensors and the reading frequency may vary if temperature readings in an area increase).

TABLE I
EXAMPLES OF COMMON REQUIREMENTS OF CONTEXT-AWARE APPLICATIONS

DISTRIBUTION AND NODE HETEROGENEITY
<ul style="list-style-type: none"> • Prediction of volcanic eruptions: Sensors placed in the crater. Machine learning algorithms for data analysis. • Smart irrigation systems: Sensors for soil and air moisture monitoring. Drones with cameras. Processing within the farm.
TIMING REQUIREMENTS
<ul style="list-style-type: none"> • Healthcare monitoring systems: Biomedical signals-with different measurement rates. • Fire detection systems: Temperature and humidity sensors with different dynamic properties must be jointly processed.
ADAPTABILITY
<ul style="list-style-type: none"> • Early warning systems: Temporal, spatial and numerical resolution of active sensors according to the current criticality level. • Nursing homes: Activation of remote monitoring of vital signs to provide emergency teams with useful information.
SCALABILITY
<ul style="list-style-type: none"> • Nursing homes: Admission or discharge of the elderly. Variations in the health status of residents. • Early warning systems: Variations in the number of sensors according to the criticality level.
SECURITY AND PRIVACY
<ul style="list-style-type: none"> • Healthcare monitoring systems: Private access to medical data. • Smart factories: Management of confidential data. Injuries, deaths and money loss due to security lacks.
SERVICE AVAILABILITY
<ul style="list-style-type: none"> • Smart factories: Rescheduling of manufacturing plan in case of machine failure. • Healthcare monitoring systems: Avoiding dangerous situations for the patient.

Resources in which services are performed may also depend on the domain. Healthcare monitoring usually demands variable processing capabilities, connection to biophysical sensors or more complex sensors such as cameras. However, in FMS, specific manufacturing assets are required, such as assembling robots, milling/drilling machines, or automated guided vehicles (AGVs).

From an implementation point of view, different distributed software architectures have been used to develop context-aware applications, such as component-based systems [13], multi-agent systems (MAS) [14], service-oriented architectures [15] or microservices [16]. Any implementation of a distributed software architecture meets distribution, heterogeneity, scalability and timing requirements, and can be extended with security features. They also support starting and stopping of applications, and communication among their distributed modules. Platforms built on these software architectures also offer dynamic reconfiguration mechanisms to cope with adaptability and availability requirements [17]–[26]. However, what is not so common in these platforms is the consideration of the application concept, although it is necessary when requirements affect a set of application entities. In this case, typical requirements are temporal (e.g., end to end deadline) or context-related when it is necessary to

make decisions on other applications. For instance, when a patient is being remotely monitored and some biophysical measurements exceed the established thresholds, new applications must be started to measure new biophysical variables. There are several proposals in the literature that attempt to undertake domain dependent demands but, in general, they are ad-hoc solutions and can hardly be applied in other domains. As far as the authors know, no platform covers all the requirements identified for context-aware applications.

Previous works of the authors proposed ad-hoc management platforms for context-aware applications, initially in the eHC field [27] and subsequently in the FMS domain [28]. The first platform had to be mostly redesigned to achieve the second one, as the application structure, resources and application management were completely different.

This work goes a step further, proposing an architecture for managing the execution of context-aware applications. It is based on a generic core that can be customized to concrete domains based on modeling artifacts. Specifically, the architecture contributes:

- The management of the execution of the application modules is driven by the key concept of application, understood as a set of interrelated domain modules.

- The logic of adaptation to relevant context changes is independent of the application functionality, being possible to act on applications.

The use of multi-agent technology allows decentralized decision-making by introducing intelligence within the domain application modules. Thus, the architecture supports decentralized deployment, and fast service recovery for stateful applications through negotiation mechanisms. The latter is based on multiple replica management and supports even node failure. A customization methodology is presented to cope with domain specific particularities. It bases on the use of model-driven development for application definition and on a set of provided agent templates for agent development.

The remainder of this paper is organized as follows. Section II presents the main requirements that a management platform for context-aware applications must meet as well as how they have been addressed in the literature. Section III is devoted to the core architecture, whose design is mainly focused on fulfilling flexibility requirements from an application-centric point of view. Section IV describes the methodology for adapting this core architecture to a specific domain, which is illustrated through a case study in the eHC field in Section V. Section VI assesses the performance of the proposal in comparison with the most popular tool for the orchestration of containerized applications and, finally, Section VII highlights some concluding remarks and future work.

II. RELATED WORK

The main objective of an application management platform is to ensure that applications execute as specified. As commented above, context-aware applications present common and domain specific demands from which the main requirements that a platform must meet can be derived. Table II collects these platform requirements which can be divided into two groups: operational (R1-R3), which tackle application execution; and non-operational, dealing with security (R4) and flexibility (R5-R8).

TABLE II
PLATFORM REQUIREMENTS

OPERATIONAL	
R1	Distributed execution and communication.
R2	Efficient application deployment.
R3	Life-cycle management.
SECURITY (NON-OPERATIONAL)	
R4	Security.
FLEXIBILITY (NON-OPERATIONAL)	
R5	Self-adaptability.
R6	Traceability/Self-awareness.
R7	Self-healing.
R8	Domain variability

From an operational point of view, the applications, deployed in heterogeneous devices, perform acquisition, processing and actuation tasks. The platform must enable the distributed execution of these tasks as well as the communication among them (*R1: Distributed execution and communication*). Besides, support for efficient deployment is

necessary, taking into account resource availability and application demands (*R2: Efficient application deployment*). Added to this, context-aware systems consist of a set of applications that are dynamic in number and size, each with its own timing requirements, whose startup, stop and normal operation must be controlled (*R3: Life-cycle management*).

Concerning non-operational requirements, system security requires mechanisms to assure the privacy, confidentiality, authentication and integrity of data (*R4: Security*). It is important to remark that context-aware applications are included within the so-called self-adaptive systems, so they also require “self-capabilities” to autonomously adapt to changes in their environment. This implies not only context-awareness but also self-awareness [29]. To achieve context-awareness, the platform must be endowed with mechanisms for application-driven dynamic reconfiguration that allow applications to react to relevant situations by changing their behavior (*R5: Self-adaptability*). Self-awareness implies being aware of dynamic resource availability. To that end, the platform must track both the state of the infrastructure resources and the state of applications (*R6: Traceability/Self-awareness*).

Regarding resource availability, the platform must minimize service interruptions, including failure detection and automatic service recovery, while maintaining the application state (*R7: Self-healing*).

Finally, every application domain has its particularities in terms of application specification (concepts that define applications and their relationships) and execution management, or even in terms of resource types. To draw on the great effort involved in the design and development of a management platform, it would be beneficial to have a platform customizable to different domains (*R8: Domain variability*).

The next subsections analyze the related work that addresses the requirements identified. Table III collects the analysis of the main management platforms related to the particular case of flexibility requirements identified in Table II.

A. OPERATIONAL REQUIREMENTS

Distributed software architectures consider applications as a set of modules (computational units) that run on different nodes and interact to achieve application functionality. However, module definition and module composition differ from one architecture to another. For example, in Component-Based Software Engineering (CBSE) [13], components are developed as black boxes that offer services in an application independent way. Applications are compositions of components based on their interface or following a component model. Applications based on MAS consist of intelligent and loosely-coupled software components, named agents, which are autonomous (they make decisions without direct human intervention), proactive (they have goal-directed behavior), reactive (they react to context changes) and social (they

interact among them, by co-operating or competing with each other) [14]. In Service Oriented Computing (SOC) [15], computational units are called services. Services are published by providers at repositories such as black boxes which consumers can discover and use, or even compose, creating new services. In the last years, the advent of microservice architectural style has allowed small and loosely-coupled services communicating through light-weight protocols, to be developed and deployed independently to compose highly scalable distributed applications [16]. The use of containerization technologies that enable lightweight virtualization has become de facto standard for packaging microservices in the cloud [30].

There are approaches aimed at easing the development and/or management of applications based on distributed architectural styles. They usually provide mechanisms for deploying, communicating and managing the life-cycle of

application modules. For example, this is the case of the Java Agent Development (JADE) [31], the most used implementation of the Foundation for Intelligent Physical Agents (FIPA) [32] standard for MAS; and Kubernetes [33], the most popular tool for the orchestration of containerized microservice-based applications. Kubernetes is usually combined with frameworks such as the Robot Operating System (ROS), which supports communications and allows orchestrating services among distributed nodes, mainly in the field of robotic applications [34], [35].

A management platform built over any of these approaches or built directly over a distributed software architecture, as those illustrated in Table III, directly meet R1 requirement (Distributed execution and communication) and, at least, a basic version of R2 (Efficient application deployment) and R3 (Life-cycle management).

TABLE III
COMPLIANCE WITH FLEXIBILITY (NON-OPERATIONAL) REQUIREMENTS BY EXECUTION MANAGEMENT PLATFORMS

PLATFORM	R5: SELF-ADAPTABILITY		R6: TRACEABILITY /SELF-AWARENESS		R7: SELF-HEALING			R8: DOMAIN VARIABILITY	
	App. driven	App. as target	Focus	Dynamic Model	App. unaware	State integrity	Node failure	App. concept	Generic
[25]	No	No	CC	No	---	---	---	No	Yes
ACCADA [17]	No	No	CC	No	---	---	---	No	Yes
MUSIC [18]	No	No	CC	No	No	Yes	No	No	Yes
DARE [19]	No	No	CC	Yes	Yes	Yes	No	No	Yes
[20]	No	No	CC	Yes	---	---	---	No	Yes
THOMAS/PANGAEA [21]/[22]	Restricted	Yes	OC	Yes	Yes	No	No	No	Yes
iLAND [23]	No	No	CC	Yes	Yes	No	Yes	Yes	No
DAMP [24]	Yes	Yes	AC	Yes	Yes	Yes	Yes	Yes	No
EI4MS [26]	No	No	UQC	Yes	Yes	Yes	No	No	Yes
MAS-RECON	Yes	Yes	AC	Yes	Yes	Yes	Yes	Yes	Yes

CC = Component-Centric, OC = Organization-Centric, AC = Application-Centric, UQC = User-perceived QoS-centric

B. NON-OPERATIONAL REQUIREMENTS

Distributed platforms can be extended by mechanisms which allow non-operational requirements to be met. The following subsections discuss research done in this direction.

1) SECURITY (R4)

A complete survey on mechanisms for ensuring secure access, storage, processing and transmission of data is presented in [36]. The most common solutions are Public Key Infrastructures (PKI), encryption, Secure Socket Layer (SSL),

authentication and authorization mechanisms, and blockchain. As all these can be included in a platform without affecting the application management, it has been considered out of the scope of this work.

2) SELF-ADAPTABILITY (R5)

Self-adaptation is usually based on the implementation of MAPE-K loop models (i.e., to apply feedback loops from control theory to autonomic computing) [37]. Self-adaptability is a complex task that can be divided into four phases: (1) monitoring/collection of context parameters; (2) detection of

relevant changes by means of the analysis of the collected data; (3) planning of appropriate adaptation actions to respond to the changes; (4) execution of the planned actions.

The first two phases are highly dependent on the specific context, which varies from one field to another [38]. Some research efforts focus on context specification [20], [39], while others, such as the one selected in this work, consider context monitoring as part of the application functionality.

Several techniques have been proposed to select adaptation actions in phase 3. Self-adaptability can be expressed through application variability [40], defined in terms of variation points (where a planned change can occur) and variants (options that can be selected). In [25] every component is considered as a variation point, and implementations as the corresponding variants, each related to a specific situation in the context. Other works apply rules, one of the most used solutions, since they provide an easy and automatable classification method. For example, [20] and [41] use rules to detect and classify relevant situations, reasoning the best response.

The actions to be executed at phase 4 range from fixed and ad-hoc proposals, such as simple warnings [41] or alarms triggering [42], to more flexible ones, based on dynamic reconfiguration. In the case of dynamic reconfiguration, an external entity automates and manages adaptation execution, separating adaptation logic from application logic. Dynamic reconfiguration has been applied at two levels: component and application. Most approaches work at component level, it being possible to add, remove, replace, and/or reconnect application modules. Sometimes, adaptation is restricted to external requests, as in the DARE framework [19], which limits the autonomy of applications. Other times, the platform itself is responsible for detecting context changes and selecting the component implementations that best fit a new context state, as in the MUSIC project [18], the platform in [25] and the ACCADA framework [17]. Similarly, the evolution-oriented EI4MS architecture [26] detects degradation on the user-perceived QoS and calculates an optimal evolution plan which is executed by the microservices themselves. To that end, the platform must be aware of the concrete context view. As adaptation at component-level does not cover the application concept, there have been attempts to extend dynamic reconfiguration to the application level. The THOMAS platform [21] and its successor PANGAEA [22] combine agent and service oriented technologies and allow structural organizations of agents. In this case, dynamic reconfiguration involves either the incorporation of new organizational structures, or the addition or removal of members. However, these capabilities are restricted to certain agent roles. In the iLAND middleware [23], dynamic reconfiguration consists of time-bounded re-composition of running service-based applications, and it is initiated by the middleware when applications are started or stopped. Previous works of the authors [27] and [24] go a step further, allowing components to ask for adaptation actions targeted to the whole

application. In [27], it is possible to start and stop already deployed applications. In [24], an ad-hoc solution for the eHC field deploys applications only when needed and allows modifying application configuration.

3) TRACEABILITY/SELF-AWARENESS (R6)

Some of the analyzed management platforms trace the system state to make the most suitable decisions at runtime. Most use a kind of repository, which varies from one platform to another. For instance, the DARE framework [19] maintains only the configuration map (i.e., the mapping of the components in execution to nodes), which is automatically discovered by means of gossiping techniques. Platforms in [25], [17] and [18] make runtime decisions based on adaptation models provided at design time. These models contain implementation alternatives according to different context values. The composition algorithms of the iLAND middleware [23] handle an application model annotated with QoS parameters that refer to data processing and resource needs of the application services. The system model used in EI4MS to elaborate evolution plans describes current deployment state of the system through information about the existing logical services, available cloud/edge nodes, user demands, and so on [26]. In [20], the context model is separated from the system model, but both include dynamic aspects that relate them at runtime. It is worth mentioning that all these works consider an application as a simple graph of interacting components with several realizations or implementations, except the latter work which allows a hierarchical component definition. In addition, they are all component-centric proposals that do not consider the application concept as a set of interrelated components managed as a whole. As a result, they cannot manage application-centric management to cope with application level demands.

There are approaches that attempt to define more complex application structures. In the context of MAS, agent-oriented engineering methodologies that take into account social concepts have been proposed for the so-called open MAS [43]: a dynamic set of agents, which may be provided by different developers, with self-interested behaviors. Specifically, these methodologies allow specifying agent-societies or agent-organizations composed of several agents, playing different roles, whose interactions are led through a set of rules, norms and constraints [44]. Based on the idea of agent-societies, the authors in [21] and [22] propose platforms for the runtime management of dynamic virtual organizations in open MAS. They provide facilities for agents to voluntarily enter or leave a virtual organization as well as for on-demand creation, deletion and modification of virtual organizations. However, it is precisely the self-interested nature of this type of agent which makes application-centric management impossible. The DAMP platform [24] considers an application as a unique entity in order to achieve application QoS enforcement. It provides a middleware service that allows application registration before its execution. This information,

together with monitoring of resource availability, is used at runtime to reconfigure applications when needed. Previous works of the authors also go beyond simple application graphs and consider applications as whole entities: [27] intended for reconfiguration driven by the application in the eHC domain, and [28], [45] aimed at fault tolerance in flexible manufacturing. However, the proposed application structures and, therefore, the corresponding system models, are fixed, which makes them non-customizable to domains with a different application concept.

4) SELF-HEALING (R7)

Self-healing is understood as the system's capacity to detect when a service becomes unavailable and to restore it. It can be proactive or reactive [46]. Proactive self-healing implies prevention, i.e., to detect service degradation before failure. Prevention tasks can be considered from the design phase, as a particular case of self-adaptability, and it is even possible to decide the optimal time to react. In contrast, reactive self-healing is a more challenging issue as it intervenes when the failure has already occurred, and allows recovery from sudden component or node failures that cannot be foreseen. Sometimes, it is even necessary to maintain the consistency of the application state.

The works in [18] and [47] support reactive self-healing by means of programming. However, although including specific code in application modules allows fast failure detection and recovery, the application logic must ensure its own availability. To avoid this dependency, replication strategies provided by the platform (transparent to the application) have been extensively applied [48].

Failure detection mechanisms are usually based on heartbeat messages. Two approaches are distinguished: 1) gossip: a component or a node informs of its liveness; 2) probe: an entity requires components or nodes to confirm they are still alive [49]. Some works propose a centralized management of heartbeat messages through a platform module in charge of detecting node failures. For example, the DAMP platform [24] and the iLAND middleware [23] use gossip techniques, whereas [50] is based on probe mechanisms. Another centralized proposal is presented in [51] which makes use of the application programming interface (API) of Kubernetes to monitor server events that indicate pod failures. There are also decentralized proposals that improve systems' autonomy and detection capacity. In the DARE framework [19], every node hosts a module in charge of gossiping to report possible node failures, whereas in [49] all nodes probe others' failures on their own.

Regarding failure recovery, some works propose a central entity with a global view of the whole system. This approach enables the separation of recovery logic from application logic and the most suitable decisions can be made. For instance, [24] and [23] make use of re-composition algorithms to select the best replica, whereas the architectures in [50] and [19] have a specific module to determine whether a failure can be recovered or not. The consistency of the application state is a

relevant point when recovering a failure, as it assures full-service continuity. Two main approaches can be found. On the one hand, check-pointing-based-recovery allows the rollback of the system to its most recent coherent state [49]. For this purpose, not only is it necessary to store the system state, but also the messages received between checkpoints. On the other hand, an easier and more flexible solution is to provide means to transfer and restore the application state. In [24] components periodically send their state to the platform, which stores it for its restoration in the new selected implementations. However, this causes an overhead on the platform and it is only possible for periodic components. In [51] the so-called State Controller component is integrated with Kubernetes to allow stateful service recovery of pods. Although it considers elasticity (i.e., multiple active pods offer the same service), state transfer is limited to concrete pairs of pods.

C. LITERATURE ANALYSIS CONCLUSIONS

In conclusion, to meet non-operational requirements, management platforms extend the implementation of a distributed software architecture. Dynamic reconfiguration is the best mechanism to accomplish self-adaptability and self-healing. In both cases, decentralized approaches improve system autonomy, decreasing platform overhead. However, a global vision of the whole system is needed to make decisions that best fit the needs of all running applications. As far as the authors know, management platforms usually focus on some, but not all, of the identified requirements. Additionally, there is a lack of application-centric management, as most proposals do not consider applications as an entity. And if they do, it is considering an ad-hoc and/or fixed structure. Achieving application-centric management requires the platform to be aware of the application concept of a specific domain. An ad-hoc definition of applications makes the corresponding management platform also an ad-hoc solution [24], [27], [28]. Adapting these platforms to other domains involves redesigning and/or re-implementing them, as in the case of the previous works of the authors [27], [28] or in the MASHA architecture (which was initially developed for web sites [52] and later adapted to eLearning systems [53]). Therefore, having a generic and customizable architecture would reduce or even avert the necessity for this hard work (R8: Domain variability). For this, it is essential to be able to define application structure in an abstract way.

III. MAS-RECON: A CUSTOMIZABLE AND APPLICATION-CENTRIC ARCHITECTURE

This section presents the MAS-RECON architecture (see Fig. 1): a generic and application-centric proposal which can be customized to specific domains.

In order to meet the operational requirements (R1-R3), the architecture is based on multi-agent technology, which has been widely used for the development of complex systems and allows the distribution of decision-making [14]. As depicted

in Fig. 1, without loss of generality, it has been built upon the JADE framework [31], but any other implementation of the FIPA specification [32] could be adopted. MAS-RECON allows decentralized decision-making by introducing intelligence within the domain entities (*Resource Agents* and *Application Agents* in Fig. 1). At the same time, all these entities are supervised at system level (*System Supervisory Agents* in Fig. 1). The information needed to achieve system-level supervision is stored in the *System Repository* (SR), which is managed by the *System Repository Agent* (SRA) alone. These two entities are the basis for the application-

centric management, contributing to meet R6 and R8. The *Event Agent* (EvA) focuses on R5, supporting application-driven reconfiguration. Finally, R7 is achieved through the *Health Monitor Agent* (HMA) and the *Recovery Agent* (ReA), which supervise resources and application agents for failure detection and recovery, including the case of stateful applications (those whose current execution state depends on previous ones). Finally, to tackle specific domain dependent requirements, such as elaborated mechanisms for admission control, new System Supervisory Agents might be included (*Other Agents* in Fig. 1).

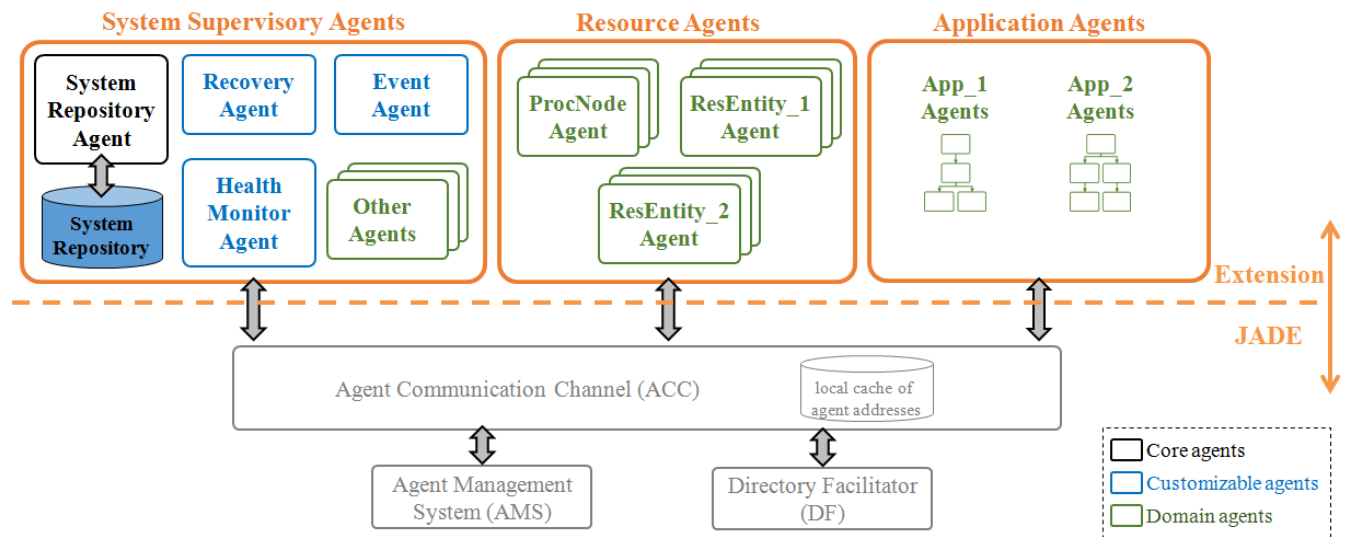


FIGURE 1. MAS-RECON architecture. It is based on an implementation of the FIPA specification to meet R1-R3 requirements. Decentralized decision-making is performed by domain entities: resources and applications, running as Resource Agents and Application Agents, respectively. System Supervisory Agents provide system-level supervision to meet different requirements: System Repository Agent focuses on R6 and R8, Event Agent on R5, and Health Monitor Agent jointly with Recovery Agent on R7. The System Repository Agent is the core of the architecture, common to all domains, whereas the other System Supervisory Agents can be customized to address domain particularities.

The following subsections describe the MAS-RECON architecture as well as the mechanisms used to meet the requirements.

A. ARCHITECTURE CORE (R6 and R8)

One of the contributions of this work is the architecture core common to all domains, which consists of the SR and the SRA, which are responsible for maintaining the state of the complete system, understood as the relevant information to tackle traceability/self-awareness (R6) and domain variability (R8).

The SR is a model that represents the system state from an application-centric management point of view. The meta-model of the SR is depicted in Fig. 2. It is closely related to the core architecture presented in Fig. 1. It contains information related to the domain entities depicted in Fig. 1 (*Resource* and *Application Agents*). This information comprises properties needed for application management, common to all domains (e.g., *id* and *agentState*), and domain dependent properties (marked as *domainProperties* in Fig. 2). These latter are determined when application concept is

defined, as it is detailed in Section IV for the particular case of eHC.

The SR collects the set of Resource Agents (*Resource Agents* in Fig. 1 and *ResourceAgent* in Fig. 2), which represent available resources, and which are characterized by the services they offer to applications (*Service* in Fig. 2). Resource entity types are enumerated in the SR (*TResource* in Fig. 2): the processing node, which is the unique resource entity common to all domains as it hosts Application Agents and System Supervisory Agents (*ProcNode* in Fig. 2 and *ProcNode Agent* in Fig. 1), as well as those related to concrete domains (for instance, *ResEntity_1* and *ResEntity_2* in Fig. 2, and *ResEntity_1 Agent* and *ResEntity_2 Agent* in Fig. 1).

The SR also collects the set of applications that can be executed (*domainApplications* in Fig. 2). Applications are defined as a set of entities (*AppEntity* in Fig. 2) that are interrelated according to the application structure defined for the specific domain. Hierarchical and/or dependency relationships might exist among them. Applications must be registered before requesting their execution. At runtime, every registered application entity has at least one associated agent

(AppAgent in Fig. 2 that corresponds to an Application Agent in Fig. 1). This will be explained in Section III.D, when discussing how MAS-RECON manages application availability. Thus, the state of the application is defined by the state of its corresponding Application Agents.

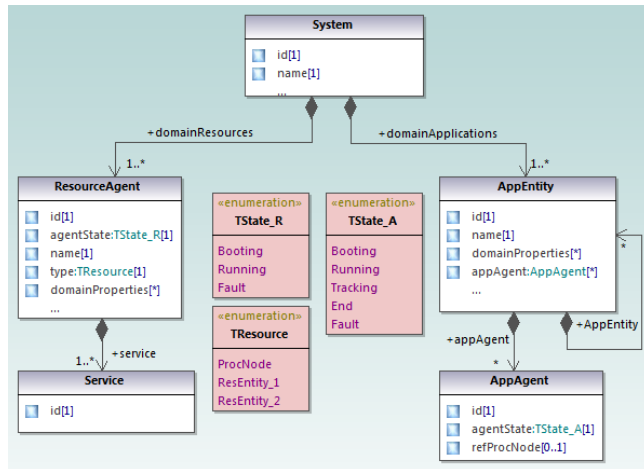


FIGURE 2. Generic structure of the System Repository. The meta-model identifies the domain entities that compose the SR (resources and applications), the relations among them, and their characterization. Hierarchical relationships among application entities are represented by the AppEntity composition links. All elements have a unique identifier assigned by the platform (id). In the case of the services offered by resource entities, their semantics are known only to domain entities, being transparent to the platform.

The SRA provides a unique access point to the SR through the generic API shown in Fig. 3.a. It allows the registration of resources and applications (*iRegAgent* and *iRegApplication* interfaces, respectively), the starting and stopping of applications (*iExecManagement* interface), and application information management (*iSystemInfo* interface).

B. OPERATIONAL REQUIREMENTS (R1-R3)

Meeting operational requirements covers the starting, stopping and normal operation of applications, taking into account that MAS-RECON relies on multi-agent technology and that the system state is stored at the SR.

Applications must be registered before being started (*Application Registration* in Fig. 3.b). The use of meta-modeling techniques to define the SR allows the SRA to handle a generic registration process that assures that applications conform to the application structure defined for the domain (*iRegApplication* interface in Fig. 3.a). Application registration is carried out in two phases. The initial phase consists of the iterative and unitary registration of all the entities that compose the application (*RegAppEntity* method in Fig. 3.a), one by one following a top-down order, according to the application hierarchy. The second phase involves the validation of the fully registered application (*AppValidation* method in Fig. 3.a). The so-called *Launcher Agent* in Fig. 3.b represents a domain-specific System Supervisory Agent that provides external users with access to

application management (it belongs to the *Other Agents* group of Fig. 1).

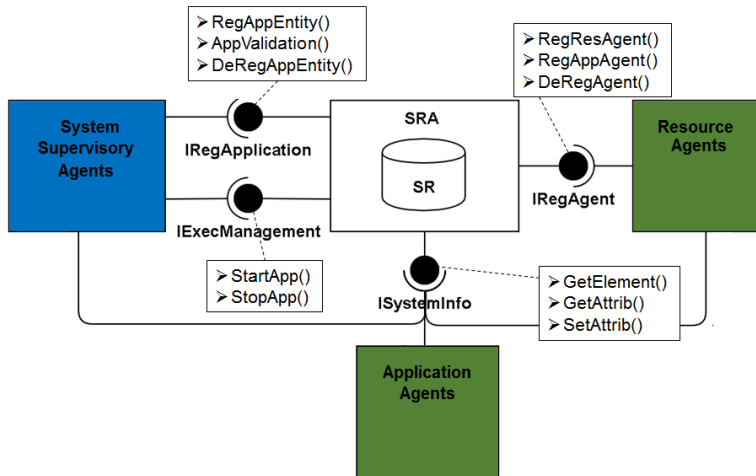
Resource entities register their corresponding Resource Agents when they are booted (*Resources Startup* in Fig. 3.b) by means of the *RegResAgent* method of the *iRegAgent* interface. Fig. 4.a presents the state diagram of the finite state machine (FSM) that describes the generic behavior of Resource Agents. The architecture provides the Resource Agent code-skeleton that implements this FSM. Once started, Resource Agents perform two tasks. On the one hand, they supervise the related physical resources (e.g., processing nodes can monitor available free memory). On the other hand, they are provided with negotiation mechanisms to decide, in a distributed way, the most suitable resource to perform a task according to specific criteria.

The *IExecManagement* interface offered by the SRA (see Fig. 3.a) allows starting and stopping applications. Starting an application implies the registration, instantiation and deployment of all the Application Agents related to its registered entities. A top-down process is proposed, divided into two phases:

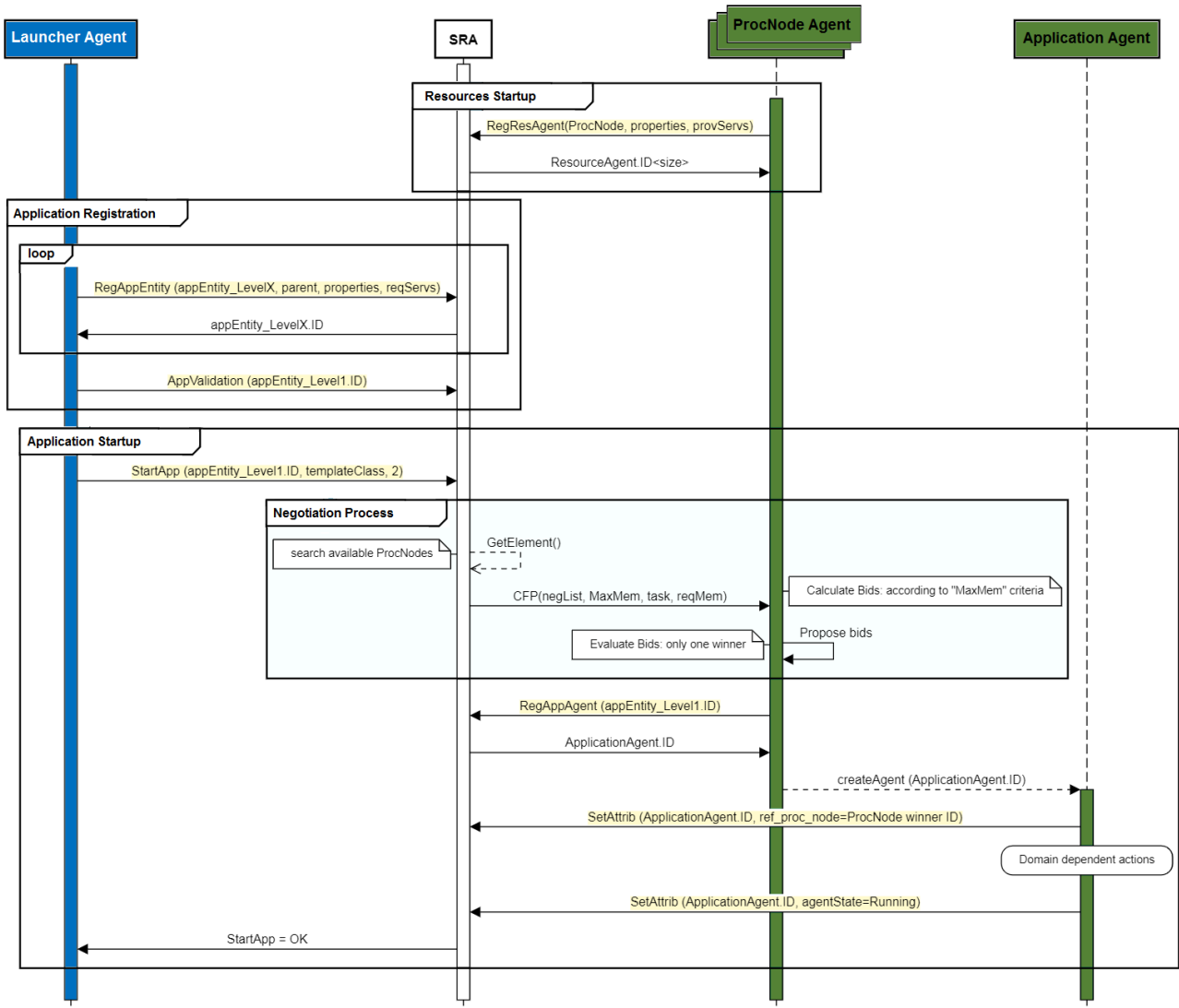
- Phase 1 (*Application Startup* in Fig. 3.b): the generic startup of first-level application entities. It is initiated upon the invocation of the *StartApp* method in Fig. 3.a. The SRA looks for the processing nodes offering the required services and launches a negotiation process among their corresponding Resource Agents (*Negotiation Process* in Fig. 3.b), including: agent data, negotiation criteria, and actions to be executed by the winner. In the example of Fig. 3.b: the memory required by the new agent and the class that implements it as agent data, the maximum free memory as negotiation criterion, and as winner actions: to register (*RegAppAgent* method), create and deploy the corresponding Application Agent (*createAgent* in Fig. 3.b).
- Phase 2 (*Domain dependent actions* in Fig. 3.b): the subsequent startup of lower-level entities in a decentralized way. Each Application Agent performs the startup of those at the next lower level. Being dependent on the concrete structure of the application, it is a domain-specific phase (see Section IV).

The FSM depicted in Fig. 4.b presents the generic behavior of Application Agents, implemented on the Application Agent code-skeleton, also provided by the architecture. At booting, they update the SR with the processing node in which they have been deployed (*refProcNode* property of the *AppAgent* element in Fig. 2) and start lower-level entities. Once booted, they execute their piece of application functionality until stopped.

Application stopping is requested through the *StopApp* method of the *IExecManagement* interface and follows the reverse process. Lower-level Application Agents deregister themselves (*DeRegAgent* method in Fig. 3.a) and stop in a down-top sequence.



a) Generic API of the System Repository Agent



b) Sequence diagram for the startup of resource and application entities

FIGURE 3. Definition and use of the generic API of the System Repository Agent (color palette: core agent in white; customizable System Supervisory Agents in blue; Resource and Application Agents in green). The figure at the top presents the interfaces defined for registering application entities and Domain Agents, starting and stopping applications, and getting or updating the information collected at the SR. The figure at the bottom describes the startup process of resource and application entities, through the use of this API (highlighted in yellow).

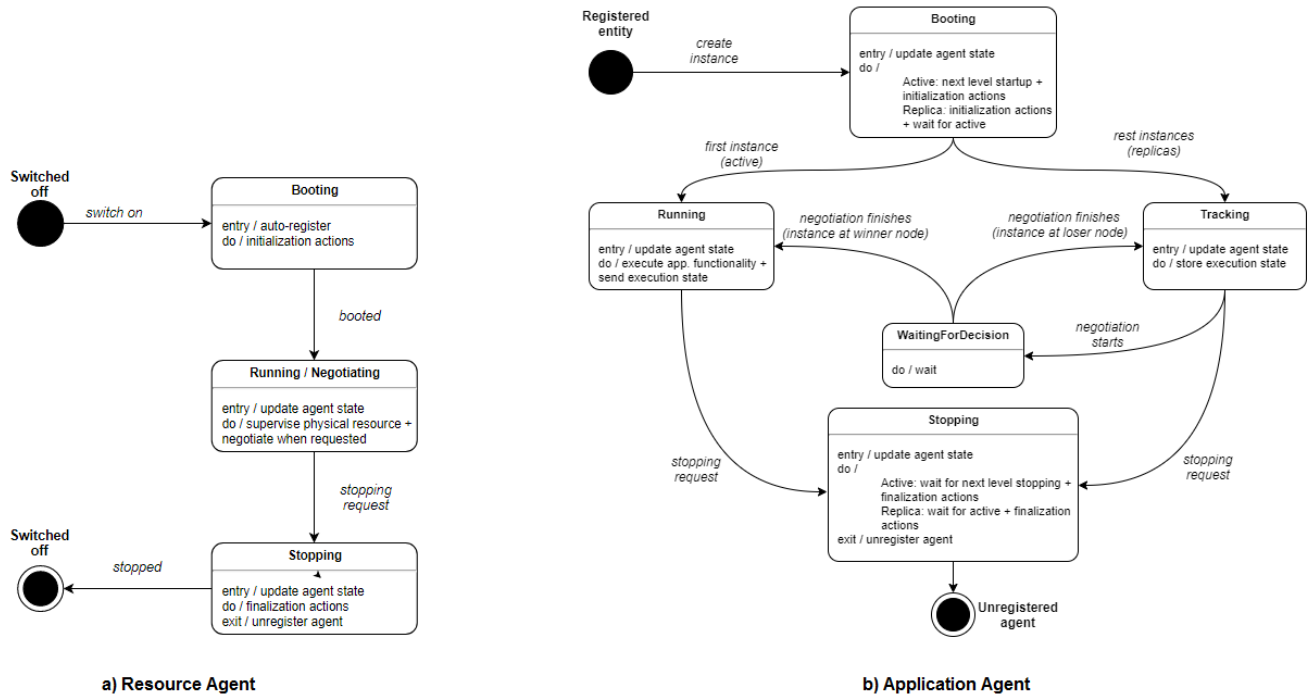


FIGURE 4. State diagram for the generic behavior of Domain Agents, implemented as the agent code-skeletons provided by the architecture: a) Resource Agents; b) Application Agents.

C. SELF-ADAPTABILITY (R5)

MAS-RECON contributes an application-driven reconfiguration approach, which is based on the notion of MAPE-K loop models [37] and handles two concepts: (1) *Event*: it identifies a relevant context change; and (2) *Action*: the reaction to an event consists of executing a set of actions, each one targeted to an application (itself or another one), which might even follow a concrete execution order. To make application logic independent of adaptation logic, events and actions must be declared during application registration, so that the EvA (see Fig. 1) performs a centralized supervision of the adaptation process. For this, the *IEvent* and *IAction* interfaces depicted in Fig. 5 have been defined.

As events and actions are domain-specific, they are specified in the domain application meta-model. Section IV describes how to adapt the characterization of these elements to a specific domain.

MAS-RECON assumes that the first two phases of self-adaptation are part of the application functionality. Specifically, they are performed by the Application Agents in charge of acquiring and processing context data (*Event Trigger Application Agent* in Fig. 5). Therefore, context particularities are unknown by the platform. The EvA is provided with the *IEvent* interface, which allows these Application Agents to report on detected events.

Then, the EvA searches the SR for the corresponding actions, and is responsible for launching and supervising their execution, through the *IAction* interface implemented by other

Application Agents (*Action Performer Application Agents* in Fig. 5).

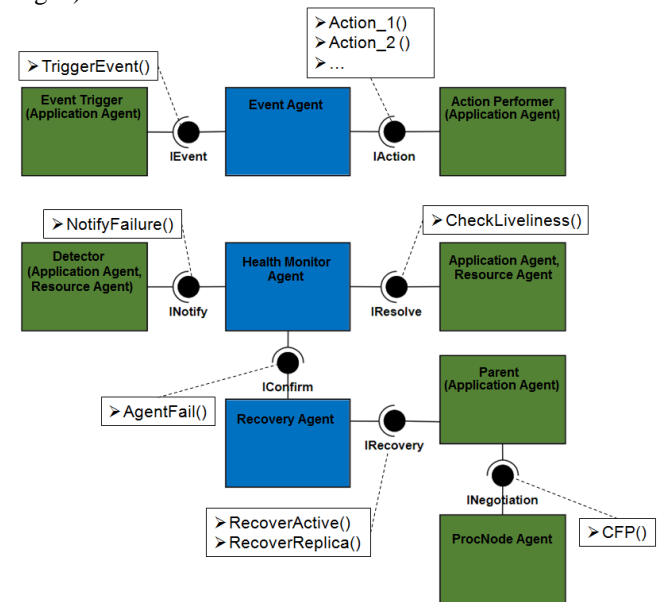


FIGURE 5. Interface definition for interactions among System Supervisory Agents and Resource and Application Agents, related to flexibility requirements: self-adaptability (R5) and self-healing (R7) (color palette: customizable System Supervisory Agents in blue; Resource and Application Agents in green).

D. REACTIVE SELF-HEALING (R7)

Following the idea of decentralized decision-making and system-level supervision, MAS-RECON supports reactive self-healing by distributed failure detection (Resource and

Application Agents) with centralized verification and recovery supervision (HMA and ReA). For this, the *INotify*, *IResolve*, *IConfirm* and *IRecovery* interfaces depicted in Fig. 5 have been defined. It abstracts the design of application functionality from self-healing mechanisms. Besides, it covers isolated agent failures and crashes at processing resources

which affect several application entities belonging to different applications. However, failures at domain specific resources are not covered by the architecture. MAS-RECON defines the sequence of messages from failure detection to recovery (summarized in Fig. 6), which determines the interactions among agents through the interfaces depicted in Fig. 5.

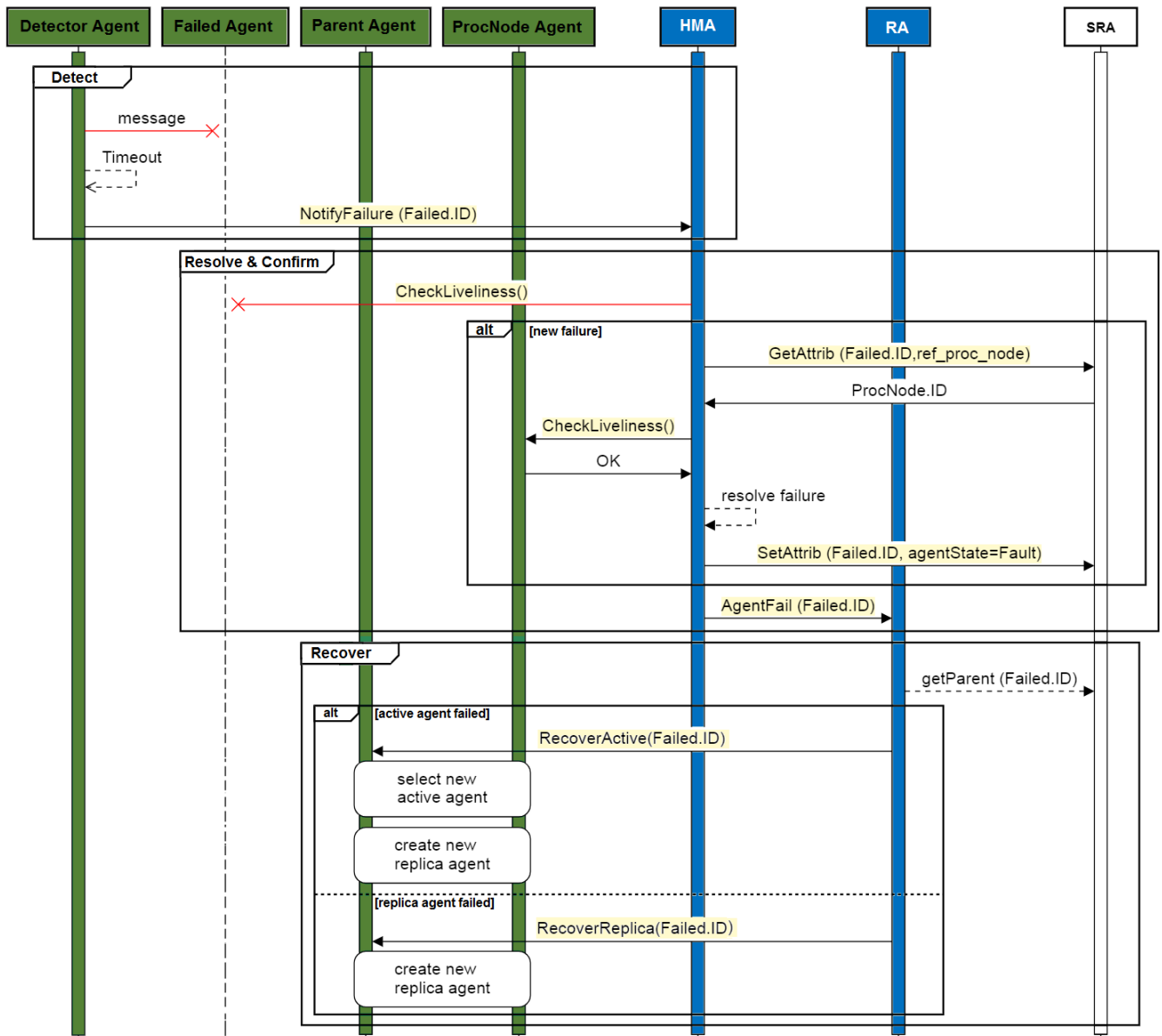


FIGURE 6. Reactive self-healing in MAS-RECON (color palette: core agent in white; customizable System Supervisory Agents in blue; Resource and Application Agents in green). The sequence diagram starts with the detection of the failure of an Application Agent (Failed Agent). It also describes the message sequence managed by the HMA to resolve and confirm the possible failure. Finally, the recovery process of confirmed failures is supervised by the ReA. The use of the API provided by the SRA is highlighted in yellow.

Failure detection relies on the potential of underlying MAS framework (e.g., JADE) to report on the non-delivery of a message (*Detect* in Fig. 6). It is undertaken by Resource and Applications Agents (identified as *Detector Agent* in Fig. 5 and Fig. 6), in the same way for all domains, through the *INotify* interface of HMA (see Fig. 5). As it is generic, the code

for failure detection is included as part of the agent code-skeleton provided by MAS-RECON.

Considering that several domain agents can detect the same failure, the HMA is the sole receptor of all the notifications, and it performs a centralized verification of the failure (*Resolve & Confirm* in Fig. 6). This includes resolving

A. STEP 1: DOMAIN SPECIFICATION

The use of a model-based approach for domain specification allows the SRA to provide a generic registration process and a generic management of the SR. The proposed approach involves the definition of the domain meta-model that determines the structure, rules and restrictions that the SR must follow [54]. Specifically, the meta-model of the SR (see Fig. 2) is implemented as an eXtensible Markup Language (XML) Schema (XSD) [55] (SR.xsd), which is composed by two other schemas:

- The *Concepts* schema defines the domain concepts. It identifies the domain entity-types: resource entities (*TResource* enumeration in Fig. 2) and application entities (*AppEntity* in Fig. 2). It also identifies their relevant characteristics, from their management point of view (*domainProperties* of *ResourceAgent* and *AppEntity* in Fig. 2). For this, XML elements and attributes are used, respectively. As adaptation actions are domain dependent, this schema also includes Event and Action concepts.
- The *Hierarchy* schema states the allowed relationships among concepts (relations between *AppEntity* elements in Fig. 2). Application hierarchy is defined through “parent-child” elements (composition of *AppEntity* elements in Fig. 2), whereas “Key/Keyref” constructs are used for dependency constraints.

The schema for SR (SR.xsd) extends the Hierarchy schema with properties common to all domains and used by MAS-RECON to fulfill all the requirements previously identified.

These schemas are used during the registration process of applications (*Application Registration* in Fig. 3.b). At the initial phase, every application entity is validated against the Concepts schema to assure its correctness, by means of the *RegAppEntity* method of Fig. 3.b. Then, the completely registered application is validated against the Hierarchy schema to ensure that it is well-formed, by means of the *AppValidation* method of Fig. 3.b. A detailed description of the registration process and its validation algorithm is found in [56].

Fig. 8 depicts the Concepts and Hierarchy schemas related to the person-centric eHC applications illustrated in Fig. 7. The *Patient* is considered the first-level application entity-type of the domain, which groups together a set of *eHC Activities* for medical supervision and actuation, according to its health status. The *eHC Activities* may be divided into several *Tasks*, which cooperate among themselves through data exchange. Most *Tasks* are related to acquisition, processing and warning or storing assignments, as in the case of the *Continuous Glucose Monitoring* eHC Activity. Although very simple *Tasks* have been represented in the case study, they can refer to complex ones, such as interpreting medical images. Apart from these hierarchical relationships, dependency relationships also exist (highlighted in orange in Fig. 7), which represent the actions to perform because of an event trigger.

Three action types are distinguished, all targeted to eHC Activities:

- Create: to initiate the execution of a new eHC Activity.
- Destroy: to finish the execution of an already running eHC Activity.
- Update: to change the properties of an eHC Activity (e.g., period or risk of level).

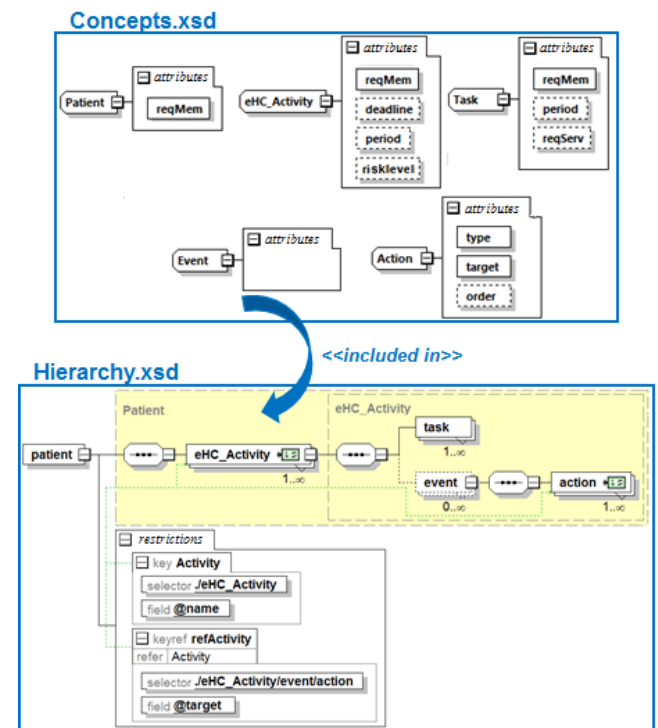


FIGURE 8. Concepts and Hierarchy XML schemas related to the person-centric applications of the eHC case study. Concepts.xsd identifies and characterizes application entities, in terms of XML elements and attributes, respectively. Hierarchy.xsd states their relationships, hierarchy (composition) and dependency (key-keyref construct).

Regarding the characterization of application entities, all are described by the memory needed by their corresponding agent, as Application Agents are deployed according to the “maximum free memory” criterion (*reqMem* in Fig. 7 and Fig. 8). Some eHC Activities are periodic (e.g., checking if *P2* Patient has relaxed before measuring their blood pressure, carried out every six hours), whereas others can execute on different risk levels (e.g., when monitoring the heart rate of *P3* Patient two risk levels can be distinguished: low (L) if acquired rate is inside its normal boundaries, or high (H) if it is out of range). Similarly, some *Tasks* run periodically. In this case, this period is different from that of eHC Activity. For instance, once *Check Relaxed* eHC Activity is activated, heart rate is acquired every 10 seconds.

Domain resource entities must also be considered. In this eHC system only processing resources, whose type is already defined in the core architecture, namely *ProcNode* type (see Fig. 2), are needed. Services offered represent the accessibility to a concrete biomedical sensor (e.g., pulsioximeter,

glucometer...). Services may be required by Task entities (e.g., *Heart Rate Acq* or *Glucose Acq* in Fig. 7).

B. STEP 2: DOMAIN AGENT TEMPLATES

Resource or Application Agents related to the same domain entity-type share the same management particularities. Therefore, an agent template can be developed for each domain entity-type (application and resource) that has a runtime representation, from which concrete Resource and Application Agents are derived.

The starting point is the agent skeleton-codes provided by MAS-RECON, which implements the FSMs of Fig. 4. Template development implies extending every FSM state according to the interactions defined among domain agents and System Supervisory Agents. The interactions established by the architecture must at least be considered (those interfaces depicted in Fig. 5 that domain agents use and/or provide). To that end, MAS-RECON also provides the code of those mechanisms for which the message sequence is fixed, namely:

- Failure detection and failure notification of non-delivered messages.
- Recovery of active instance or replicas.
- Negotiation process, including both launching a new negotiation process (*CFP* in Fig. 3), and sending and evaluating bids.

In the case of Resource Agents (see Fig. 4.a), required initialization and finalization actions will be considered at *Booting* and *Stopping* FSM states, respectively. *Running/Negotiation* state comprises the supervision of the concrete physical resource (e.g., the amount of free memory at processing nodes or the amount of remaining battery charge in AGVs). It is also the place to implement concrete negotiation mechanisms (e.g., the largest available memory in processing nodes or the time needed to cover a distance in AGVs).

Regarding Application Agents (see Fig. 4.b), *Booting* and *Stopping* FSM states are related to the application startup and stop, respectively. As MAS-RECON itself covers the startup of the first-level application entities (*Application Startup* in Fig. 3.b), application agent templates focus on the subsequent booting of the entities at lower levels, level by level until the last one is reached, according to the concrete application concept. This includes:

- 1) Looking for next-level entities (child), by means of queries to the SR through the SRA.
- 2) Launching a negotiation process among processing resources for every child, considering the required services.
- 3) Waiting for child agents to be started, except in the case of last-level entities.
- 4) Informing upper-level entity (parent) that booting has finished.

The implementation of the *Running* and *Tracking* FSM states focuses on the normal execution and on flexibility needs. For self-adaptability, it is necessary to identify, at least,

which entities are in charge of detecting relevant context changes (*Event Trigger Application Agents* in Fig. 5), and which are responsible for executing adaptation actions (*Action Performer Application Agents* in Fig. 5). The former make use of the IEvent interface, whereas the latter implement the IAction interface. Similarly, for self-healing, at least, the following agents have to be identified: those in charge of failure detection (*Detector Resource* and *Application Agents* in Fig. 5), and those which execute recovery actions (*Parent Application Agents* in Fig. 5).

In the eHC case study, no resource templates are needed, since there are no domain specific resources (note that the ProcNode agent template is part of MAS-RECON). However, three templates are necessary for Patient, eHC Activity and Task agents. As an example, Fig. 9 represents the functions tackled by Task Agents at each FSM state, and which have to be implemented in the corresponding template. The code provided by MAS-RECON is marked in black whereas domain dependent code is highlighted in blue. According to the application structure depicted in Fig. 8, during application startup (*Booting* FSM state), Patient Agents supervise the creation of eHC Activity Agents and these latter do the same with Task Agents. As Task Agents exchange data messages among them, it is necessary to synchronize their start-up. Regarding self-adaptability, Task Agents are Trigger Application Agents because they process context data, being able to detect context changes. As all actions are targeted at eHC Activities, Patient Agents are the Action Performer Application Agents in charge of creating, destroying or updating eHC Activities. Finally, as far as self-healing is concerned, all Application Agents act as Detector Application and Resource Agents as they communicate through messages. Parent Application Agents are determined according to the specified application hierarchy.

C. STEP 3: EXTENSION OF SYSTEM SUPERVISORY AGENTS

MAS-RECON implements the interfaces provided by the System Supervisory Agents that represent application management needs common to all domains (see Fig. 5). Domain specific needs can be tackled in two ways. On the one hand, it is possible to extend these common interfaces to consider concrete adaptability actions or new ways of failure detection and/or recovery. On the other hand, new System Supervisory Agents could be included in the architecture. Their interfaces and the corresponding implementation derive from the analysis of interactions with other agents, which should also be included in the domain agent templates. In the eHC case study, a new System Supervisory Agent called Launcher Agent has been added to handle the external requests for application registration, start and stop.

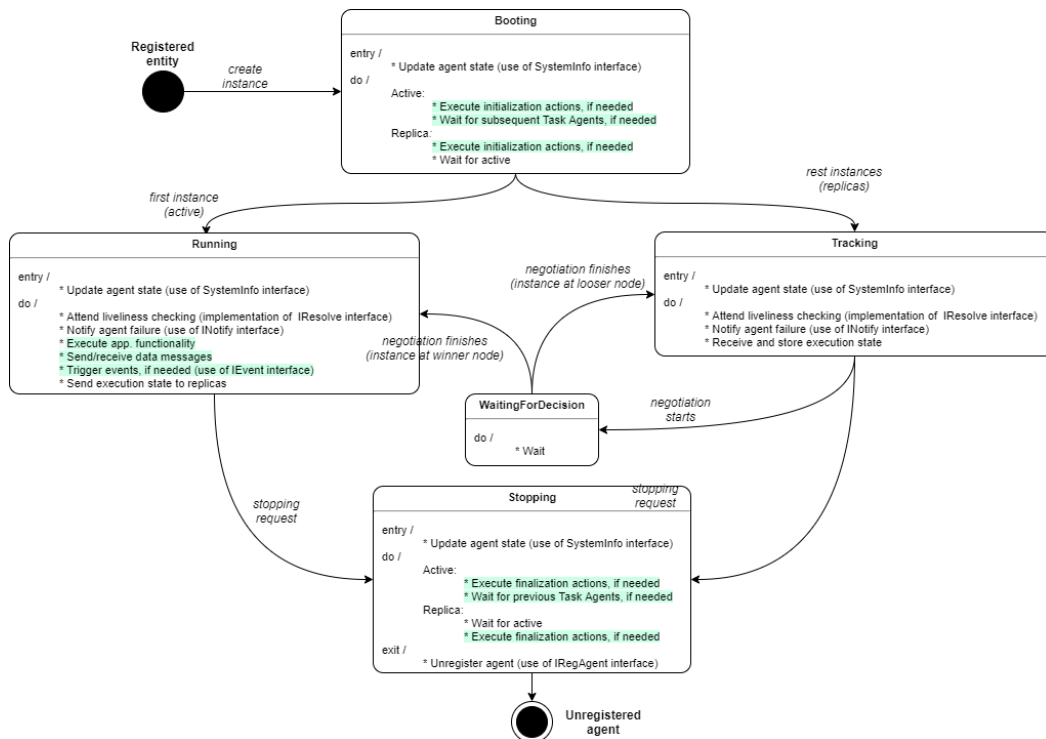


FIGURE 9. Customization needs for the development of the template of Task Agents in eHC. Functionalities tackled by Task Agents at every FSM state are depicted. Those implemented by the code provided by MAS-RECON are marked in black (e.g., notify agent failure). Those implemented in domain dependent code are highlighted in blue (e.g., implementation of the logic for triggering events related to relevant context changes, through the IEvent interface): Note that Tracking and Waiting for Decision FSM states do not have domain dependent functionality.

V. CASE STUDY: A PLATFORM FOR THE eHC DOMAIN

This section validates that the management platform built for the eHC domain, based on the MAS-RECON architecture and following the proposed customization methodology covers the requirements in Table II.

Two main tests have been carried out. The first one focuses on testing the fulfillment of operational requirements (R1-R3) and self-adaptability (R5) for a specific domain (R8). The second one evaluates self-healing on node failure (R7) and self-awareness mechanisms (R6).

From an infrastructure point of view, the test bed consists of the following resources:

- 2 Raspberry Pi (Node_2 and Node_3), with access to biomedical sensors through the so-called “e-Health Sensor Platform V2.0.” shield [57]. They measure the blood glucose level of P1 Patient (Gluc_P1 service of Node_2) and the heart rate and blood pressure of P2 Patient (Pulsioxy_P2 and Sphyg_P2 services of Node_3), respectively.
- 4 PC (Node_1, Node_4-Node_6), only to host agents.

From a software point of view, the functionality related to Tasks has been implemented as a Java library whose methods are invoked at the Running and Tracking FSM states of Task Agents.

Once the platform is launched on Node_1 (i.e., when all System Supervisory Agents are initiated), the SRA creates the

SR, which is initially empty. Then, the other processing nodes are also booted, registering themselves at the SR as described in Section III.B.

A. APPLICATION REGISTRATION AND START-UP

The first test starts with the registration of the P2 Patient application. All application entities are registered one by one, following the application hierarchy depicted in Fig. 8: 1) Patients; 2) eHC Activities; 3) Tasks; 4) Events; 5) Actions. The SRA assigns a unique identifier to every registered entity (*id* attribute in Fig. 2). Finally, the correctness of the whole application is validated. This model-based registration process can prevent errors such as incorrect properties for application entities (e.g., registering a Patient entity without reqMem property, which does not match the Concepts schema in Fig. 8) or incorrect parent-child relationships (e.g., registering a Task entity as a Patient’s child, which does not match the Hierarchy schema in Fig. 8).

The SRA initiates the startup of the P2 Patient application as in Fig. 3.b. An excerpt of the negotiation process carried out to deploy the active instance of P2 Patient Agent is presented in Fig. 10. The ProcNode Agent related to Node_4 is the negotiation winner, as its bid is the best one. On the contrary, when the ProcNode Agent related to Node_5 receives a proposal better than its bid, it considers itself a loser, and leaves the negotiation process. Then, as represented in Fig. 11, the active instance of the P2 Patient Agent is responsible for the subsequent startup of its lower-level entities. Within its

Boot FSM state, it reads, from the SR, the eHC Activities to create and start two agent instances for every one (replication factor is 1 in this case): one is the active instance and the other the replica. This process is repeated level by level, until Task Agents are created.

The start-up results in the deployment presented in Fig. 12, where containers, depicted by green directories, represent processing nodes. Each container hosts the corresponding Resource Agent and those Application Agent instances whose negotiation has won. For example, Node_4 contains its Resource Agent (*procno103* id in Fig. 12), a Patient Agent

instance (*patien102* id in Fig. 12) and two Task Agent instances (*task102* and *task106* ids in Fig. 12).

Note that the container of Node_3 hosts only a Task Agent instance. The reason is twofold. On the one hand, Heart Rate Acq Task is the only application entity that requires the Pulsioxy_P2 service. Thus, it must be allocated on Node_3. On the other hand, Node_3 does not win negotiations for other instances (Patient Agents, eHC Activity Agents or other Task Agents) as its memory availability is less than that of PCs. This latter is also the reason why Node_2 does not host an application agent instance.

```

Node_4
12:43:48.786 [procno103] INFO NegotiatingBehaviour action 179 - msg=negotiate procno104,procno101,procno105,procno102,procno103 crite
rion=max mem action=start patient102 externaldata=patient102,patient,es.ehu.domain.orion2030.templates.PatientTemplate,running,4MB
12:43:48.786 [procno103] INFO NegotiatingBehaviour initNegotiation 301 - Negotiation(id:1043) procno103(value:458257392) Bid
12:43:49.818 [procno103] INFO NegotiatingBehaviour action 206 - Negotiation(id:1043) proposal procno101(value:114564344)
12:43:49.820 [procno103] INFO BasicFunctionality checkNegotiation 99 - negotiation(id:1043) partial winner procno103(value:458257392)
12:43:49.830 [procno103] INFO NegotiatingBehaviour action 206 - Negotiation(id:1043) proposal procno105(value:458257272)
12:43:49.834 [procno103] INFO BasicFunctionality checkNegotiation 99 - negotiation(id:1043) partial winner procno103(value:458257392)
12:44:45.958 [procno103] INFO NegotiatingBehaviour action 206 - Negotiation(id:1043) proposal procno102(value:114566112)
12:44:45.959 [procno103] INFO BasicFunctionality checkNegotiation 99 - negotiation(id:1043) partial winner procno103(value:458257392)
12:44:46.975 [procno103] INFO NegotiatingBehaviour action 206 - Negotiation(id:1043) proposal procno104(value:458257133)
12:47:48.275 [procno103] INFO BasicFunctionality checkNegotiation 99 - negotiation(id:1043) partial winner procno103(value:458257392)
12:47:48.276 [procno103] INFO NegotiatingBehaviour action 237 - procno103 WON negotiation(id:1043)!

Node_5
12:43:44.651 [reg-28342] INFO ResourceBootBehaviour action 101 - reg-28342: autoreg >
procno104: es.ehu.platform.template.ResourceAgentTemplate.setup()
cmd=del reg-28342
12:43:48.785 [procno104] INFO NegotiatingBehaviour action 179 - msg=negotiate procno104,procno101,procno105,procno102,procno103 crite
rion=max mem action=start patient102 externaldata=patient102,patient,es.ehu.domain.orion2030.templates.PatientTemplate,running,4MB
12:43:48.786 [procno104] INFO NegotiatingBehaviour initNegotiation 301 - Negotiation(id:1043) procno104(value:458257133) Bid
12:43:48.801 [procno104] INFO NegotiatingBehaviour action 206 - Negotiation(id:1043) proposal procno103(value:458257392)
12:43:48.806 [procno104] INFO NegotiatingBehaviour action 226 - Negotiation(id:1043) loser procno104(value:458257133) dropped

```

FIGURE 10. Example of a negotiation process carried out during the start-up of P2 Patient application. It corresponds to the deployment of the active instance of P2 Patient entity. As it does not require any concrete service all available ProcNode Agents negotiate ('procno104' id is Node_5; 'procno101' id is Node_3; 'procno105' id is Node_6; 'procno102' id is Node_2; and 'procno103' id is Node_4). Node_4 is the winner as its bid is the best one. When a ProcNode Agent receives a better bid, it assumes that it cannot win and leaves the negotiation. This is the case of Node_5.

B. EVENT MANAGEMENT

Once started, P2 patient application is executed as illustrated in Fig. 13. The main objective of the application is to monitor the blood pressure of P2 patient. However, to avoid so-called "white coat syndrome", it is not measured until the patient is relaxed [58]. To that end, the P2 Patient application has been defined as two eHC Activities (Check Relaxed and Blood Pressure Measuring) which are related through an event that represents patient relaxation.

Patient relaxation is monitored by periodically acquiring heart rate values of P2 patient. (*Step 1* and *Step 2* in Fig. 13). These measurements are also stored for further processing (*Step 3* in Fig. 13). These steps are implemented by the Task entities that compose the Check Relaxed eHC Activity. When relaxation is detected (*Step 4* in Fig. 13), the Relaxed Event is triggered, leading to the interactions and message sequence depicted in *Step 5*. Each event triggers actions that may affect any registered application entity. In this case, one application entity (Check Relaxed eHC Activity) is stopped whereas another application entity (the Blood Pressure Measuring eHC Activity) is initiated. This latter captures systolic and diastolic

blood pressure (*Step 6* in Fig. 13), which are also stored for further processing (*Step 7* in Fig. 13).

C. FAILURE RECOVERY

The second test assesses the ability of MAS-RECON to recover from node failures. In this test, 20 Patient applications similar to P1 Patient are registered and started, with replication factor stated as 1. When all the applications are running, the fail of Node_5 is forced (a PC that hosts Application Agents without required services). Fig. 14 depicts the evolution of memory use of the processing nodes described above, from the start-up of the applications to failure recovery.

As observed, initially (*Instant 1* in Fig. 14), those Task Agents that require the Glucometer_P1 service are deployed to Node_2. Node_3 does not hold an agent instance, since no application entity requires its services and has less memory than PCs. The rest of Application Agents are distributed in a balanced way, according to their memory needs.

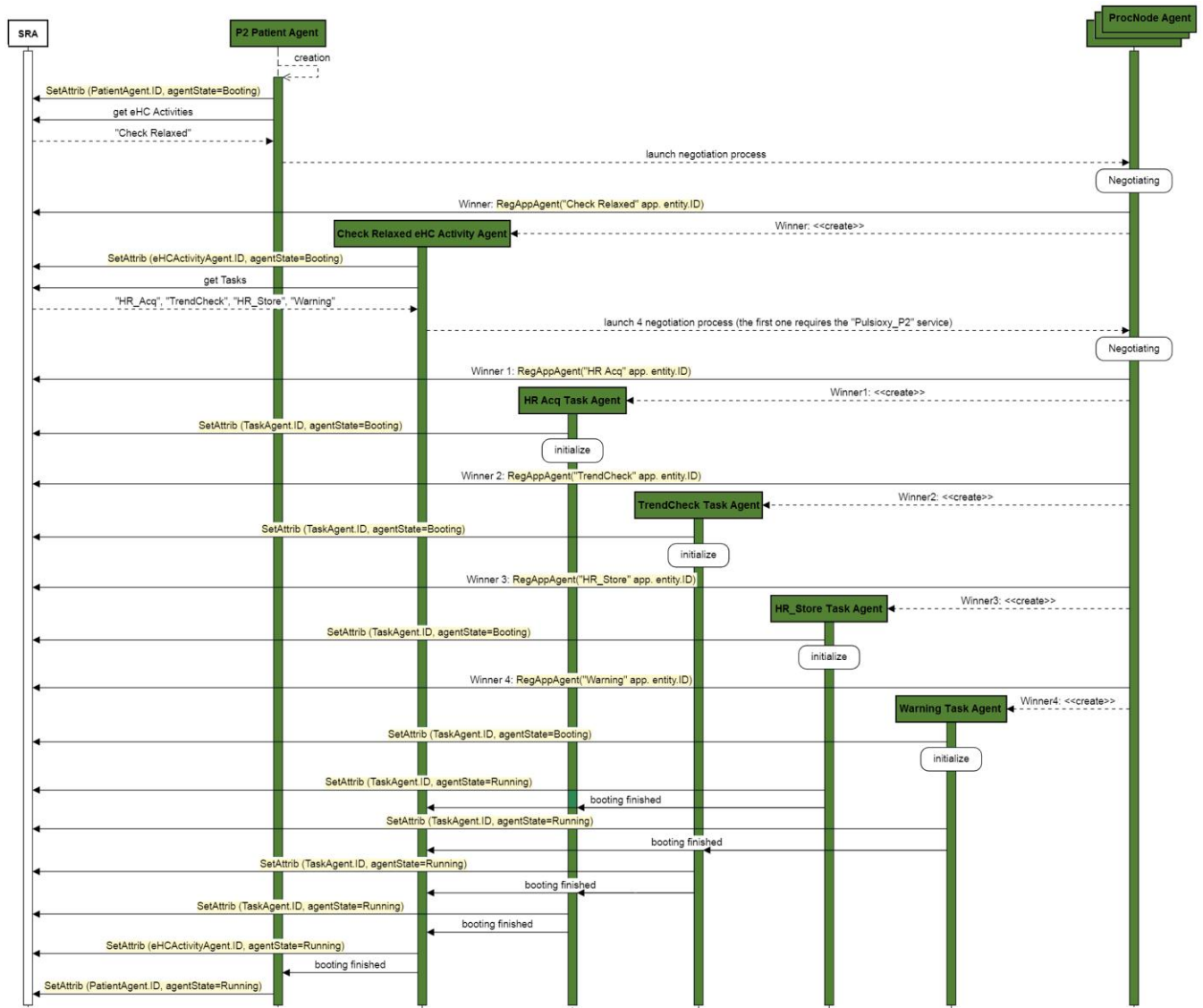


FIGURE 11. Startup sequence for P2 Patient application (color palette: core agent in white; Resource and Application Agents in green). For simplicity, redundancy level has not been considered and Heart Rate has been abbreviated as HR. Although P2 Patient consists of two eHC Activities, Blood Pressure Measuring is not initially deployed because it is event-triggered. To achieve synchronization among Task Agents, when their booting is finished, they warn the corresponding eHC Activity Agent and other Task Agents with which they communicate. The use of the API provided by the SRA is highlighted in yellow.

When the HMA receives the notification of the failure of Node_5 (*Instant 2* in Fig. 14), it verifies it and identifies the affected agents. Finally, it reports the ReA that supervises the failure recovery as follows (by looking up the information stored at the SR):

- Firstly, instances of Patient Agents are re-instantiated by the ReA itself, also supervising the necessary negotiation processes for failed active instances.
- Secondly, the ReA asks the active instance of Patient Agents to tackle the recovery of failed eHC Activity Agents, including the supervision of negotiation processes for failed active instances.
- Finally, the ReA asks the active instance of eHC Activity Agents to deal with the recovery of failed Task

Agents, including the supervision of negotiation processes for failed active instances

After the recovery (*Instant 3* in Fig. 14), the memory use of Node_4 and Node_6 increases in a balanced way. However, Node_2 and Node_3 are not affected (their memory use does not change), because they have limited resources and do not win any negotiation processes.

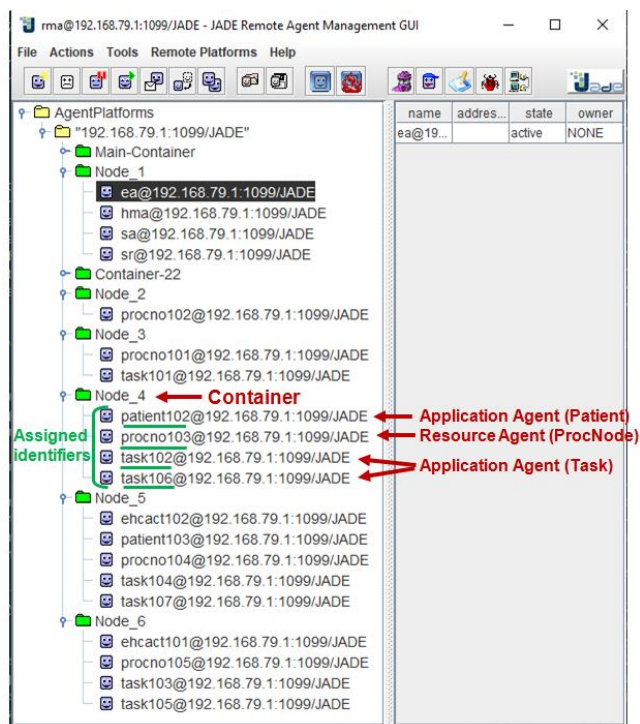


FIGURE 12. Initial deployment of P2 Patient application. Processing nodes are represented by containers. Each container hosts its corresponding Resource Agent and several Application Agents, according to the result of the negotiation processes carried out during the start-up.

VI. PERFORMANCE ANALYSIS

This section analyzes the performance of MAS-RECON, so that developers working with industry standard platforms for microservices can appreciate its benefits. The objective of this analysis is twofold: to benchmark deployment times of MAS-RECON against other application management architectures available on the market, and to extend the failure recovery analysis of Section V.C with response time measurements.

From an infrastructure point of view, the test bed for both performance analysis consists of 1 PC (Dell Precision 3551 with Intel Core i9-10885H and 64GB of RAM) that hosted a cluster of virtual machines (3 CPU and 3.5GB of RAM) created with multipass. Both the host and the virtual machines use Ubuntu 20.04 operative system.

A. BENCHMARK OF DEPLOYMENT TIMES: MAS-RECON VS. KUBERNETES

Kubernetes was selected as the industry standard implementation of microservices management platform against which to compare MAS-RECON. K3s, a lightweight and easy to install Kubernetes distribution, was selected to build the Kubernetes cluster.

Deployed applications were composed of three modules: a Generator that produces a pair of random numbers that are sent to a Processor that adds them up and sends the result to a Consumer that prints it in the standard output. The functionality of each module was programmed in Java and encapsulated in a Docker container in the case of Kubernetes, and in an agent in the case of MAS-RECON. In Kubernetes, each container was deployed using one-container-per-pod model, the three containers were related one to each other to form the application through a docker-compose file. In order for the comparison to be made on the same term, in MAS-RECON a very simple application structure was defined, consisting only of a first-level application entity-type called Component. Replication factor was stated to 0.

Two main tests were carried out. The first one focused on testing deployment times in MAS-RECON and Kubernetes for different workloads in a cluster made up of a fixed number of $N=20$ nodes and over a maximum workload of $30*N=600$ modules (components from now on). In the second test, the same measurements were taken for a 100% workload ($30*N$ components) in different cluster sizes. System Supervisory Agents of MAS-RECON and the control-plane of Kubernetes were deployed on the host machine, whereas processing nodes were installed on the virtual machines.

Measures were taken with a gateway agent that collected the timestamps of agent events in the case of MAS-RECON, and with a watcher used to listen to pod events in the case of Kubernetes. The initial timestamp in each test is the deployment request time.

Fig. 15 shows the mean scheduling, creation and startup times of both platforms for 10% (60 components), 25% (150 components), 50% (300 components), 75% (450 components) and 100% (600 components) workloads in a cluster made up of $N=20$ nodes. The results show that: 1) Kubernetes schedules components ~ 1.5 times faster than MAS-RECON; 2) MAS-RECON creates components ~ 8.3 times faster than Kubernetes; and 3) Kubernetes starts components faster than MAS-RECON on low workloads, but startup times converge at 100% workload and the trendlines suggest that MAS-RECON starts components faster than Kubernetes on higher workloads.

Fig. 16 depicts the mean scheduling, creation and startup times of both platforms for $N=1$, $N=5$, $N=10$, $N=15$, $N=20$ cluster sizes for a 100% workload ($30*N$ components). The results related to scheduling and creation times resemble those obtained in the previous test. Regarding startup times, Kubernetes starts components faster than MAS-RECON at small cluster sizes, but startup times converge at $N=20$ nodes and the trendlines suggest that MAS-RECON starts components faster than Kubernetes on bigger clusters.

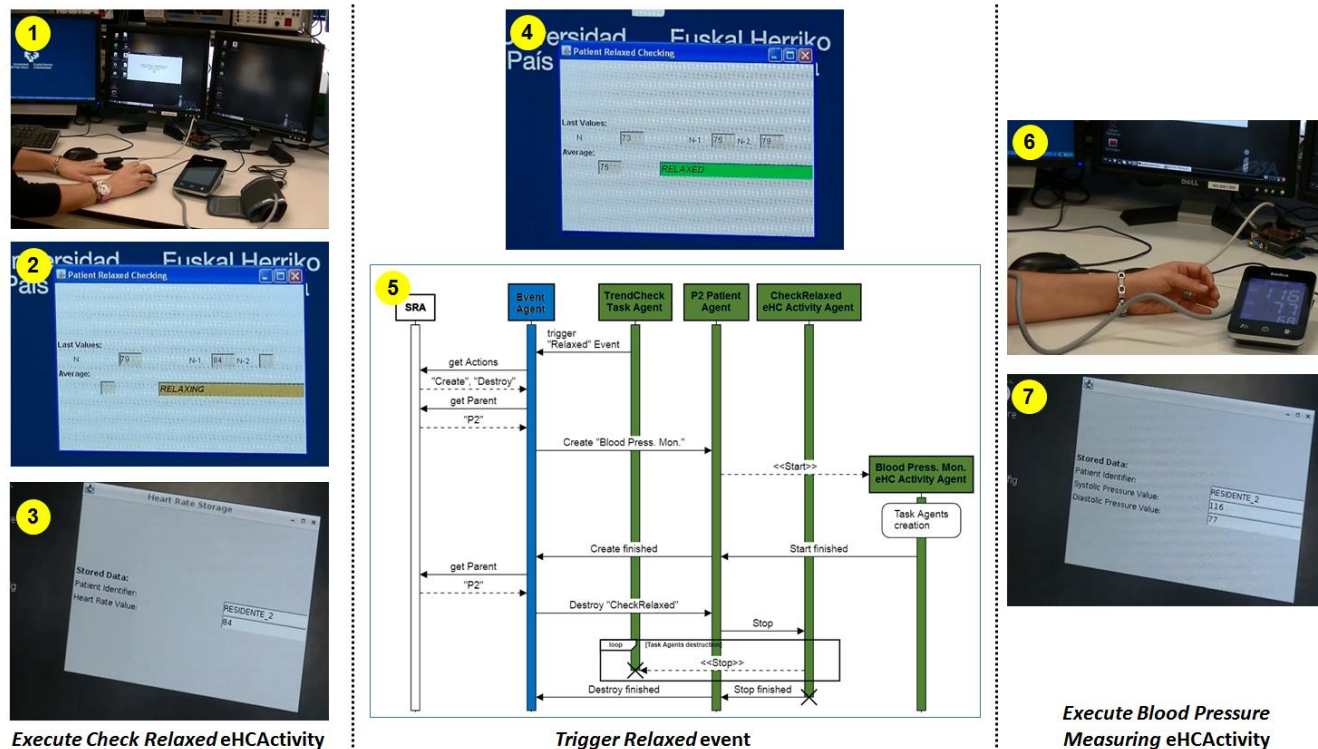


FIGURE 13. Execution steps of P2 Patient application. Initially, *Check Relaxed* eHC Activity is executed. When the relaxation of P2 Patient is detected by the *TrendCheck* Task Agent, the corresponding *Relaxed* Event is triggered, resulting in the measurement of its blood pressure (initialization of *Blood Pressure Measuring* eHC Activity) and finishing its heart rate supervision (stopping *Check Relaxed* eHC Activity).

The observed differences in planning times can be attributed to two reasons. On the one hand, the negotiation mechanism used in MAS-RECON to distribute the scheduling decision among the processing nodes is based on an adaptation of the Contract-Net protocol, which is not optimized for this task. On the other hand, in MAS-RECON the scheduling of Component agents was synchronized, as they exchange message data as in the case of Task Agents in eHC: first, the Generator is planned and, when it is created, the Processor is planned; then, when the latter is created, the Consumer is planned. Since the initial timestamp is the same for all the components of the deployment, this synchronized startup leads to higher scheduling times. It should be remarked that this synchronization cannot be achieved in Kubernetes without customizing it.

Regarding differences in creation times, the creation of an agent is faster than starting a container, since the former involves instantiation of a Java class, whereas the latter involves the instantiation of a virtual machine image.

B. RESPONSE TIME FOR FAILURE RECOVERY

This probe focused on testing agent recovery times in MAS-RECON for applications of different size in a case of a node failure. The test was performed in a cluster made up of 8 nodes where the same application structure described in the previous section was maintained. But in this case a Generator component, that produces a pair of random numbers, was connected to N Processor components in serial that increase their input in one unit. The last Processor component was

connected with a Consumer component that prints the result in the standard output. Again, connection among components was based on message exchange, and replication factor was stated to 0. Processor components were restricted to be deployed on nodes 2-7 (six nodes), whereas Generator and Consumer components were deployed on nodes 1 and 8. The failure involves the disconnection of one of the nodes and the recovery of all the failed components.

To reflect the different aspects to be considered in the recovery process, the following measurements were collected: reaction time, repair time and recovery time [51]. The reaction time measures the time elapsed from the failure until the platform starts to respond (i.e., until the HMA receives notification of the failure). The repair time measures the time elapsed from the detection of the failure until the first failed component is recovered. Finally, the recovery time measures the time elapsed from the detection of the failure until the application is restored (i.e., all failed components are recovered and the Consumer component prints a valid result again).

Table IV shows the reaction, repair and recovery times for applications made up of N=60 (recovery of 10 components), N=150 (recovery of 25 components), N=300 (recovery of 50 components), N=450 (recovery of 75 components) and N=600 (recovery of 100 components) Processor components.

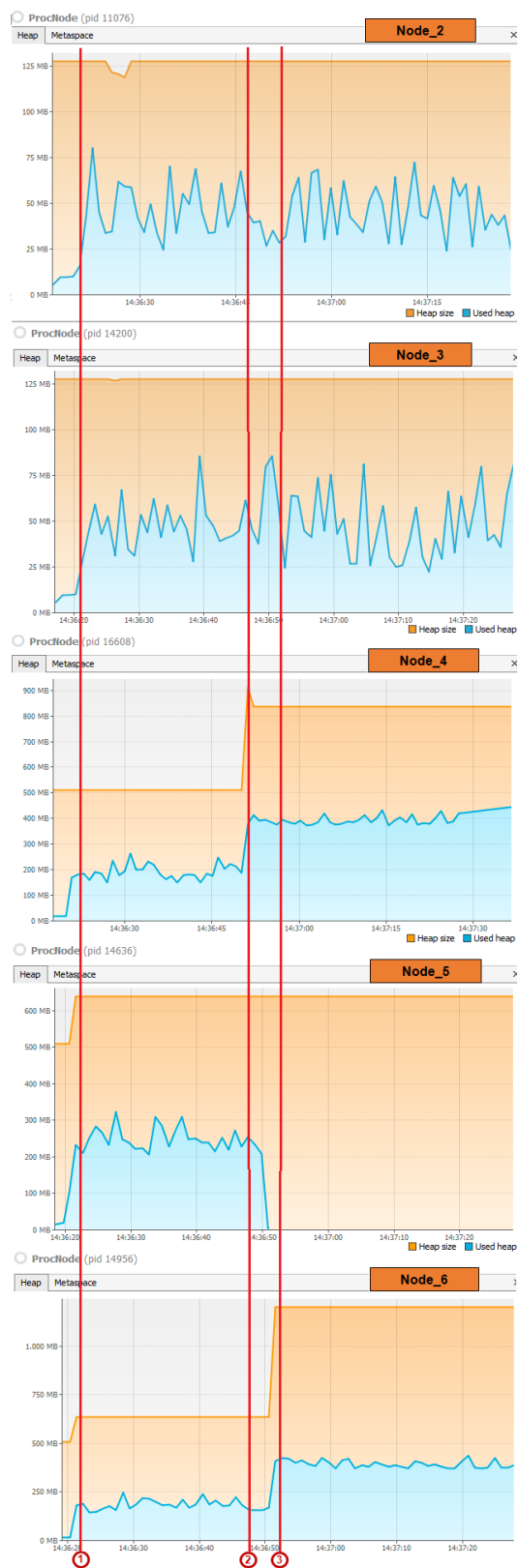


FIGURE 14. Recovery process of Node 5. The figure represents the evolution of memory use during the failure detection and recovery. Instant 1 represents the initial memory distribution after start-up. Instant 2 refers to the failure of Node 5. Finally, Instant 3 indicates the result of the recovery process.

The results show that the reaction and repair times remain approximately constant in order of magnitude (they suffer an increase of less than 0.70s between the lightest and the heaviest load). These results were expected, since the reaction time is a load-independent capability of the platform; in turn, the repair time is measured for the first component recovered, which makes it also independent of the number of components to be recovered. Finally, the recovery time increases, as expected, with the number of components to be recovered, but the ratio between the recovery time and the number of recovered components is ~ 1 (i.e., it takes approximately ~ 1 s to recover a component).

TABLE IV

RESULTS OF RESPONSE TIMES FOR FAILURE RECOVERY WITH APPLICATIONS OF DIFFERENT SIZE (N). FAILURE RECOVERY IS BASED ON THE FAILURE OF ONE NODE (I.E., N/6 COMPONENTS ARE LOST)

	N=60	N=150	N=300	N=450	N=600
Reaction	3,14s	3,14s	3,14s	3,54s	3,81s
Repair	5,28s	5,56s	5,33s	4,35s	5,97s
Recovery	8,38s	18,62s	52,87s	85,81s	96,16s

VI. CONCLUSIONS

This paper has proposed MAS-RECON, a generic and customizable architecture for the management of context-aware applications. It is mainly focused on considering the domain application concept from its initial design, also fulfilling the operational and flexibility requirements of target applications from an application-centric point of view.

The formalization of the domain based on models facilitates platform customization, allowing a generic management of the system state. It has also been proven that distributed intelligence, (achieved through multi-agent technology) jointly with system-level supervision, makes it possible to face unexpected events: namely, relevant context changes or agent failures. The MAS-RECON architecture, together with the proposed customization methodology, allows domain specific platforms to be developed, which meet common and domain dependent requirements of context-aware applications.

From the analysis of deployment times of MAS-RECON against Kubernetes, it can be concluded that scheduling time in Kubernetes is better than in MAS-RECON. This is mainly due to the personalization facilities that MAS-RECON offers, which, among others, allow startup or deployment of agents customized to concrete domains. In fact, it can be said that MAS-RECON goes beyond other management platforms, being a kind of development framework that eases agent implementation through the definition of templates.

However, the approach still has limitations. Currently, MAS-RECON lacks an admission control, which assures that applications are accepted only if there are enough resources. Additionally, given the dynamism of resource availability, flexible QoS management is needed to adjust the QoS level of running applications to the available resources at any moment.

Another serious drawback is the effort to develop a particular platform, as it requires an in-depth knowledge of the MAS-RECON architecture to integrate a new code. In this sense, future work is aimed at extending MAS-RECON architecture

to facilitate the development of new domain platforms. Thus, by making the most of model-driven engineering in terms of model transformations, it will be possible to customize the architecture to different domains.

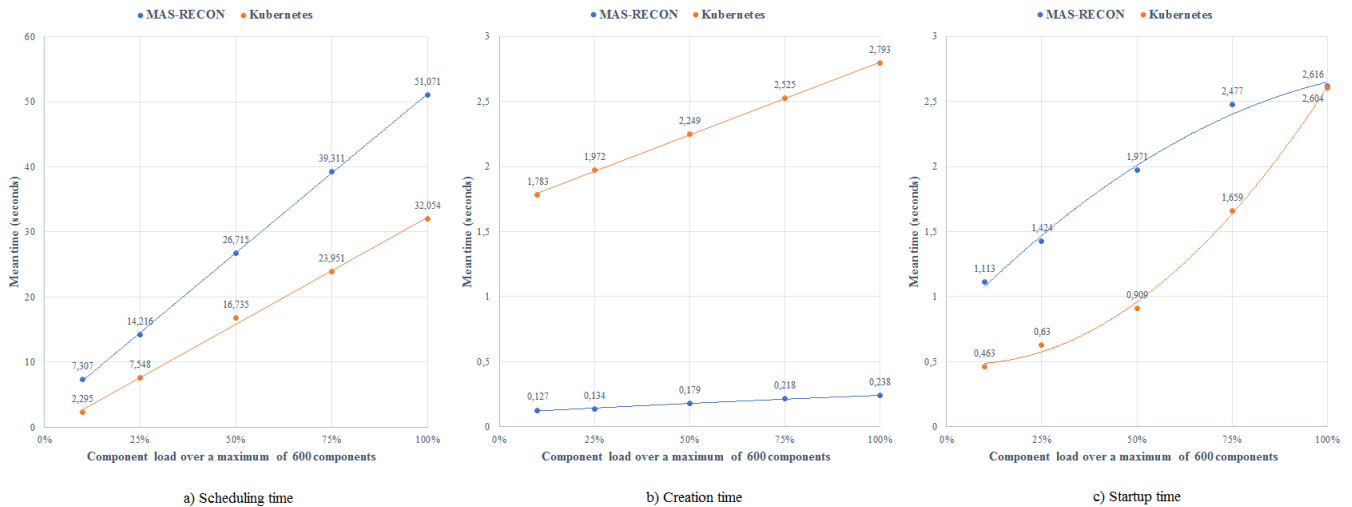


FIGURE 15. Component deployment times in MAS-RECON (blue color) and Kubernetes (orange color), for different workloads in a cluster made up of a fixed number of N=20 nodes and over a maximum workload of 30*N (600) components: a) mean scheduling time; b) mean creation time; c) mean startup time.

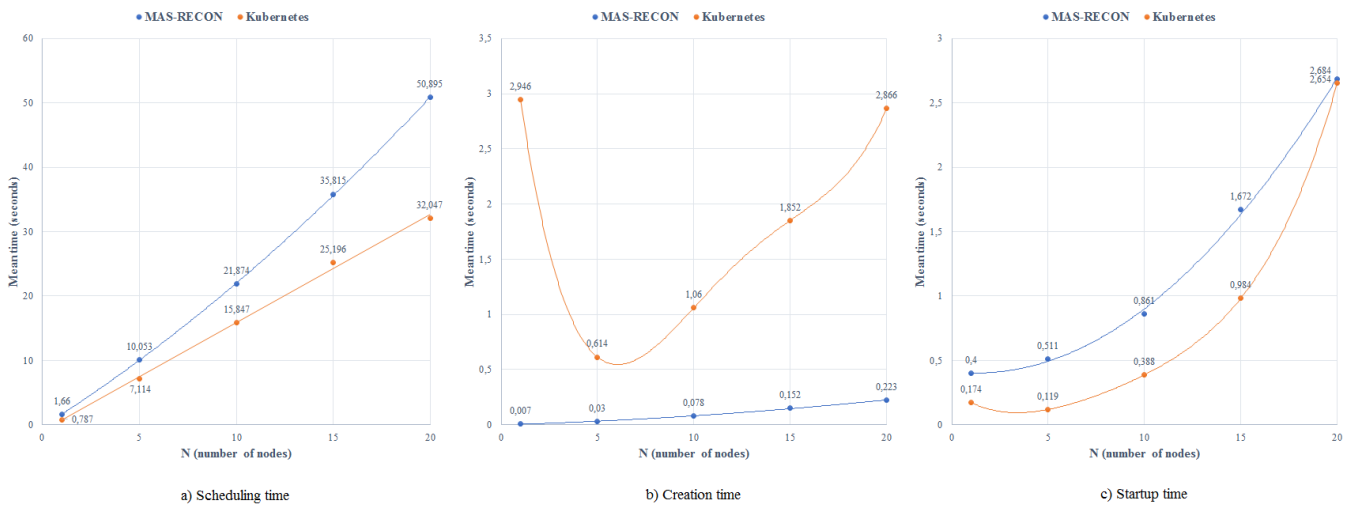


FIGURE 16. Component deployment times in MAS-RECON (blue color) and Kubernetes (orange color), for a 100% workload (30*N components) in different cluster sizes: a) mean scheduling time; b) mean creation time; c) mean startup time.

REFERENCES

[1] A. Čolaković and M. Hadžialić, "Internet of Things (IoT): A review of enabling technologies, challenges, and open research issues," *Comput. Netw.*, vol. 144, pp. 17–39, 2018, doi: 10.1016/j.comnet.2018.07.017.

[2] C. Perera, C. H. Liu, S. Jayawardena, and M. Chen, "A Survey on Internet of Things From Industrial Market Perspective," vol. 2, pp. 1660–1679, 2014, doi: 10.1109/ACCESS.2015.2389854.

[3] H. Boyes, B. Hallaq, J. Cunningham, and T. Watson, "The industrial internet of things (IIoT): An analysis framework," *Comput. Ind.*, vol. 101, pp. 1–12, 2018, doi: 10.1016/j.compind.2018.04.015.

[4] H. Xu, W. Yu, D. Griffith, and N. Golmie, "A Survey on Industrial Internet of Things: A Cyber-Physical Systems Perspective," *IEEE Access*, vol. 6, pp. 78238–78259, 2018, doi: 10.1109/ACCESS.2018.2884906.

[5] P. P. Ray, M. Mukherjee, and L. Shu, "Internet of Things for Disaster Management: State-of-the-Art and Prospects," *IEEE Access*, vol. 5, pp. 18818–18835, 2017, doi: 10.1109/ACCESS.2017.2752174.

[6] S. A. Shah, D. Z. Seker, M. M. Rathore, S. Hameed, S. Ben Yahia, and D. Draheim, "Towards Disaster Resilient Smart Cities: Can Internet of Things and Big Data Analytics Be the Game Changers?," *IEEE Access*, vol. 7, pp. 91885–91903, 2019, doi: 10.1109/ACCESS.2019.2928233.

[7] S. B. Baker, W. Xiang, and I. Atkinson, "Internet of Things for Smart Healthcare: Technologies, Challenges, and Opportunities," *IEEE Access*, vol. 5, pp. 26521–26544, 2017, doi: 10.1109/ACCESS.2017.2775180.

- [8] S. M. R. Islam, D. Kwak, M. H. Kabir, M. Hossain, and K.-S. Kwak, "The Internet of Things for Health Care: A Comprehensive Survey," *IEEE Access*, vol. 3, pp. 678–708, 2015, doi: 10.1109/ACCESS.2015.2437951.
- [9] M. Ayaz, M. Ammad-Uddin, Z. Sharif, A. Mansour, and E.-H. M. Aggoune, "Internet-of-Things (IoT)-Based Smart Agriculture: Toward Making the Fields Talk," *IEEE Access*, vol. 7, pp. 129551–129583, 2019, doi: 10.1109/ACCESS.2019.2932609.
- [10] M. S. Farooq, S. Riaz, A. Abid, K. Abid, and M. A. Naeem, "A Survey on the Role of IoT in Agriculture for the Implementation of Smart Farming," *IEEE Access*, vol. 7, pp. 156237–156271, 2019, doi: 10.1109/ACCESS.2019.2949703.
- [11] B. Chen, J. Wan, L. Shu, P. Li, M. Mukherjee, and B. Yin, "Smart Factory of Industry 4.0: Key Technologies, Application Case, and Challenges," *IEEE Access*, vol. 6, pp. 6505–6519, 2018, doi: 10.1109/ACCESS.2017.2783682.
- [12] Q. Qi and F. Tao, "A Smart Manufacturing Service System Based on Edge Computing, Fog Computing, and Cloud Computing," *IEEE Access*, vol. 7, pp. 86769–86777, 2019, doi: 10.1109/ACCESS.2019.2923610.
- [13] T. Vale, I. Crnkovic, E. S. de Almeida, P. A. da M. Silveira Neto, Y. C. Cavalcanti, and S. R. de L. Meira, "Twenty-eight years of component-based software engineering," *J. Syst. Softw.*, vol. 111, pp. 128–148, 2016, doi: 10.1016/j.jss.2015.09.019.
- [14] Michael Wooldridge, *An Introduction to MultiAgent Systems*, 2nd ed. Hoboken, NJ, USA: Wiley Publishing, 2009.
- [15] J. Al-Jaroodi and N. Mohamed, "Service-oriented middleware: A survey," *J. Netw. Comput. Appl.*, vol. 35, no. 1, pp. 211–220, Jan. 2012, doi: 10.1016/j.jnca.2011.07.013.
- [16] James Lewis and Martin Fowler, "Microservices." 2014. Accessed: Jul. 20, 2021. [Online]. Available: <https://martinfowler.com/articles/microservices.html>
- [17] N. Gui, V. De Florio, H. Sun, and C. Blondia, "Toward architecture-based context-aware deployment and adaptation," *J. Syst. Softw.*, vol. 84, no. 2, pp. 185–197, 2011, doi: 10.1016/j.jss.2010.09.017.
- [18] S. Hallsteinsen et al., "A development framework and methodology for self-adapting applications in ubiquitous computing environments," *J. Syst. Softw.*, vol. 85, no. 12, pp. 2840–2859, 2012, doi: 10.1016/j.jss.2012.07.052.
- [19] E. Albassam, J. Porter, H. Goma, and D. A. Menasce, "DARE: A Distributed Adaptation and Failure Recovery Framework for Software Systems," in 2017 IEEE International Conference on Autonomic Computing (ICAC), Columbus, OH, USA, 2017, pp. 203–208. doi: 10.1109/ICAC.2017.12.
- [20] M. Hussein, J. Han, and A. Colman, "An Approach to Model-Based Development of Context-Aware Adaptive Systems," in 2011 35th IEEE Annual Computer Software and Applications Conference, Munich, Germany, 2011, pp. 205–214. doi: 10.1109/COMPSAC.2011.34.
- [21] E. Argente, V. Botti, C. Carrascosa, A. Giret, V. Julian, and M. Rebollo, "An abstract architecture for virtual organizations: The THOMAS approach," *Knowl. Inf. Syst.*, vol. 29, no. 2, pp. 379–403, 2011, doi: 10.1007/s10115-010-0349-1.
- [22] G. Villarrubia, D. Hernández, J. F. De Paz, and J. Bajo, "Combination of multi-agent systems and embedded hardware for the monitoring and analysis of diuresis," *Int. J. Distrib. Sens. Netw.*, vol. 13, no. 7, pp. 1–17, 2017, doi: 10.1177/1550147717722154.
- [23] M. Garcia Valls, I. R. Lopez, and L. F. Villar, "iLAND: An Enhanced Middleware for Real-Time Reconfiguration of Service Oriented Distributed Real-Time Systems," *IEEE Trans. Ind. Inform.*, vol. 9, no. 1, pp. 228–236, 2013, doi: 10.1109/TII.2012.2198662.
- [24] A. Agirre, J. Parra, A. Armentia, E. Estévez, and M. Marcos, "QoS Aware Middleware Support for Dynamically Reconfigurable Component Based IoT Applications," *Int. J. Distrib. Sens. Netw.*, vol. 2016, no. Article ID 2702789, pp. 1–17, 2016, doi: 10.1155/2016/2702789.
- [25] M. U. Khan, R. Reichle, and K. Geihs, "Architectural Constraints in the Model-Driven Development of Self-Adaptive Applications," *IEEE Distrib. Syst. Online*, vol. 9, no. 7, pp. 1–10, 2008, doi: 10.1109/MDSO.2008.19.
- [26] X. He, Z. Tu, X. Xu, and Z. Wang, "Programming framework and infrastructure for self-adaptation and optimized evolution method for microservice systems in cloud-edge environments," *Future Gener. Comput. Syst.*, vol. 118, pp. 263–281, May 2021, doi: 10.1016/j.future.2021.01.008.
- [27] A. Armentia, U. Gangoiti, R. Priego, E. Estévez, and M. Marcos, "Flexibility Support for Homecare Applications Based on Models and Multi-Agent Technology," *Sensors*, vol. 15, no. 12, pp. 31939–31964, 2015, doi: 10.3390/s151229899.
- [28] M. López, J. Martín, U. Gangoiti, A. Armentia, E. Estévez, and M. Marcos, "Tolerancia a fallos en Sistema de Fabricación Flexible basado en MAS," in XXXIX Jornadas de Automática, Badajoz, Spain, 2018, pp. 799–805. doi: <https://doi.org/10.17979/spudc.9788497497565>.
- [29] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems," *Pervasive Mob. Comput.*, vol. 17, pp. 184–206, 2015, doi: 10.1016/j.pmcj.2014.09.009.
- [30] Y. Wang, H. Kadiyala, and J. Rubin, "Promises and challenges of microservices: an exploratory study," *Empir. Softw. Eng.*, vol. 26, no. 4, p. 63, Jul. 2021, doi: 10.1007/s10664-020-09910-y.
- [31] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa, "JADE: A software framework for developing multi-agent applications. Lessons learned," *Inf. Softw. Technol.*, vol. 50, no. 1–2, pp. 10–21, 2008, doi: 10.1016/j.infsof.2007.10.008.
- [32] Foundation for Intelligent Physical Agents, "Standard FIPA specifications." 2002.
- [33] "Kubernetes." 2020. Accessed: Jul. 22, 2021. [Online]. Available: <https://kubernetes.io/docs/home/>
- [34] P. González-Nalda, I. Etxeberria-Agiriano, I. Calvo, and M. C. Otero, "A modular CPS architecture design based on ROS and Docker," *Int. J. Interact. Des. Manuf. IIJDeM*, vol. 11, no. 4, pp. 949–955, Nov. 2017, doi: 10.1007/s12008-016-0313-8.
- [35] G. Toffetti, T. Lötscher, S. Kenzhegulov, J. Spillner, and T. M. Bohnert, "Cloud Robotics: SLAM and Autonomous Exploration on PaaS," in Companion Proceedings of the 10th International Conference on Utility and Cloud Computing, Austin Texas USA, Dec. 2017, pp. 65–70. doi: 10.1145/3147234.3148100.
- [36] V. Hassija, V. Chamola, V. Saxena, D. Jain, P. Goyal, and B. Sikdar, "A Survey on IoT Security: Application Areas, Security Threats, and Solution Architectures," *IEEE Access*, vol. 7, pp. 82721–82743, 2019, doi: 10.1109/ACCESS.2019.2924045.
- [37] P. Arcaini, E. Riccobene, and P. Scandurra, "Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation," in 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Florence, Italy, 2015, pp. 13–23. doi: 10.1109/SEAMS.2015.10.
- [38] X. Li, M. Eckert, J.-F. Martinez, and G. Rubio, "Context Aware Middleware Architectures: Survey and Challenges," *Sensors*, vol. 15, no. 8, pp. 20570–20607, Aug. 2015, doi: 10.3390/s150820570.
- [39] J. R. Hoyos, J. García-Molina, and J. A. Botía, "A domain-specific language for context modeling in context-aware systems," *J. Syst. Softw.*, vol. 86, no. 11, pp. 2890–2905, 2013, doi: 10.1016/j.jss.2013.07.008.
- [40] M. Galster, D. Weyns, D. Tofan, B. Michalik, and P. Avgeriou, "Variability in Software Systems—A Systematic Literature Review," *IEEE Trans. Softw. Eng.*, vol. 40, no. 3, pp. 282–306, 2014, doi: 10.1109/TSE.2013.56.
- [41] A. Rocha et al., "Innovations in health care services: The CAALYX system," *Int. J. Med. Inf.*, vol. 82, no. 11, pp. e307–e320, 2013, doi: 10.1016/j.ijmedinf.2011.03.003.
- [42] T. Wu, F. Wu, J.-M. Redoute, and M. R. Yuce, "An Autonomous Wireless Body Area Network Implementation Towards IoT Connected Healthcare Applications," *IEEE Access*, vol. 5, pp. 11413–11422, 2017, doi: 10.1109/ACCESS.2017.2716344.
- [43] L. R. Coutinho, A. A. F. Brandão, O. Boissier, and J. S. Sichman, "Towards Agent Organizations Interoperability: A Model Driven Engineering Approach," *Appl. Sci.*, vol. 9, no. 12, pp. 1–38, 2019, doi: 10.3390/app9122420.
- [44] J. J. Gómez-Sanz and R. Fuentes-Fernández, "Understanding Agent-Oriented Software Engineering methodologies," *Knowl. Eng. Rev.*, vol. 30, no. 4, pp. 375–393, 2015, doi: 10.1017/S0269888915000053.
- [45] U. Gangoiti, A. López, A. Armentia, E. Estévez, and M. Marcos, "Model-Driven Design and Development of Flexible Automated

- Production Control Configurations for Industry 4.0,” *Appl. Sci.*, vol. 11, no. 5, pp. 1–27, Mar. 2021, doi: 10.3390/app11052319.
- [46] H. Psailer and S. Dustdar, “A survey on self-healing systems: approaches and systems,” *Computing*, vol. 91, pp. 43–73, Jan. 2011, doi: 10.1007/s00607-010-0107-y.
- [47] I. García-Magariño and C. Gutiérrez, “Agent-oriented modeling and development of a system for crisis management,” *Expert Syst. Appl.*, vol. 40, no. 16, pp. 6580–6592, 2013, doi: 10.1016/j.eswa.2013.06.012.
- [48] R. Guerraoui and A. Schiper, “Software-Based Replication for Fault Tolerance,” *Computer*, vol. 30, no. 4, pp. 68–74, 1997, doi: 10.1109/2.585156.
- [49] W. Huan and N. Hidenori, “Failure Detection in P2P-Grid Environments,” in *32nd International Conference on Distributed Computing Systems Workshops*, Macau, China, 2012, pp. 369–374. doi: 10.1109/ICDCSW.2012.18.
- [50] A. Ruiz, G. Juez, P. Schleiss, and G. Weiss, “A safe generic adaptation mechanism for smart cars,” in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, Gaithersbury, MD, USA, 2015, pp. 161–171. doi: 10.1109/ISSRE.2015.7381810.
- [51] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, “A Kubernetes controller for managing the availability of elastic microservice based stateful applications,” *J. Syst. Softw.*, vol. 175, p. 110924, May 2021, doi: 10.1016/j.jss.2021.110924.
- [52] D. Rosaci and G. M. L. Sarné, “MASHA: A multi-agent system handling user and device adaptivity of Web sites,” *User Model. User-Adapt. Interact.*, vol. 16, no. 5, pp. 435–462, 2006, doi: 10.1007/s11257-006-9015-4.
- [53] S. Garruzzo, D. Rosaci, and G. M. L. Sarne, “MASHA-EL: A Multi-Agent System for Supporting Adaptive E-Learning,” in *19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2007)*, Patras, Greece, 2007, pp. 103–110. doi: 10.1109/ICTAI.2007.83.
- [54] B. Selic, “The pragmatics of model-driven development,” *IEEE Softw.*, vol. 20, no. 5, pp. 19–25, 2003, doi: 10.1109/MS.2003.1231146.
- [55] W3C, “XML Schema Part 0: Primer (Second Edition), W3C REC-xmldata-20041028.” 2004. [Online]. Available: <https://www.w3.org/TR/2004/REC-xmldata-20041028/>
- [56] Casquero, O., Armentia, A., Estevez, E., López, A., and M. Marcos, “Customization of agent-based manufacturing applications based on domain modelling,” presented at the *21st IFAC World Congress*, Berlin, Germany, 2020. Accessed: Nov. 18, 2020. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S0952197608001437>
- [57] cooking hacks, “e-Health Sensor Platform V1.0 for Arduino and Raspberry Pi [Biometric / Medical Applications].” 2013. [Online]. Available: <https://www.cooking-hacks.com/documentation/tutorials/ehealth-v1-biometric-sensor-platform-arduino-raspberry-pi-medical.html>
- [58] Á. Jobbágy, P. Csordás, and A. Mersich, “Blood Pressure Measurement at Home,” *Seoul, Korea*, 2006, vol. 14, pp. 3453–3456. doi: https://doi.org/10.1007/978-3-540-36841-0_873.