Mälardalen University Licentiate Thesis 278

Formally Assured Intelligent Systems for Enhanced Ambient Assisted Living Support

Ashalatha Kunnappilly

MÄLARDALEN UNIVERSITY
SWEDEN

Mälardalen University Press Licentiate Theses
No. 278

FORMALLY ASSURED INTELLIGENT SYSTEMS FOR ENHANCED AMBIENT ASSISTED LIVING SUPPORT

Ashalatha Kunnappilly

2019

School of Innovation, Design and Engineering

ISBN 978-91-7485-425-1
ISSN 1651-9256
Printed by E-Print AB, Stockholm, Sweden

Abstract

Ambient Assisted Living (AAL) solutions are aimed to assist the elderly in their independent and safe living. During the last decade, the AAL field has witnessed a significant development due to advancements in Information and Communication Technologies, Ubiquitous Computing and Internet of Things. However, a closer look at the existing AAL solutions shows that these improvements are used mostly to deliver one or a few functions mainly of the same type (e.g. health monitoring functions). There are comparatively fewer initiatives that integrate different kinds of AAL functionalities, such as fall detection, reminders, fire alarms, etc., besides health monitoring, into a common framework, with intelligent decision-making that can thereby offer enhanced reasoning by combining multiple events.

To address this shortage, in this thesis, we propose two different categories of AAL architecture frameworks onto which different functionalities, chosen based on user preferences, can be integrated. One of them follows a centralized approach, using an intelligent Decision Support System (DSS), and the other, follows a truly distributed approach, involving multiple intelligent agents. The centralized architecture is our initial choice, due to its ease of development by combining multiple functionalities with a centralized DSS that can assess the dependency between multiple events in real time. While easy to develop, our centralized solution suffers from the well-known single point of failure, which we remove by adding a redundant DSS. Nevertheless, the scalability, flexibility, multiple user accesses, and potential self-healing capability of the centralized solution are hard to achieve, therefore we also propose a distributed, agent-based architecture as a second solution, to provide the community with two different AAL solutions that can be applied depending on needs and available resources. Both solutions are to be used in safety-critical applications, therefore their design-time assurance, that is, providing a guarantee that they meet

functional requirements and deliver the needed quality-of-service, is beneficial.

Our first solution is a generic architecture that follows the design of many commercial AAL solutions with sensors, a data collector, DSS, security and privacy, database (DB) systems, user interfaces (UI), and cloud computing support. We represent this architecture in the Architecture Analysis and Design Language (AADL) via a set of component patterns that we propose. The advantage of using patterns is that they are easily re-usable when building specific AAL architectures. Our patterns describe the behavior of the components in the Behavioral Annex of AADL, and the error behavior in AADL's Error Annex. We also show various instantiations of our generic model that can be developed based on user requirements. To formally assure these solutions against functional, timing and reliability requirements, we show how we can employ exhaustive model checking using the state-of-art model checker, UPPAAL, and also statistical model-checking techniques with UPPAAL SMC, an extension of the UPPAAL model checker for stochastic systems, which can be employed in cases when exhaustive verification does not scale. The second proposed architecture is an agent-based architecture for AAL systems, where agents are intelligent entities capable of communicating with each other in order to decide on an action to take. Therefore, the decision support is now distributed among agents and can be used by multiple users distributed across multiple locations. Due to the fact that this solution requires describing agents and their interaction, the existing core AADL does not suffice as an architectural framework. Hence, we propose an extension to the core AADL language - The Agent Annex, with formal semantics as Stochastic Transition Systems, which allows us to specify probabilistic, non-deterministic and real-time AAL system behaviors. In order to formally assure our multi-agent system, we employ the state-of-art probabilistic model checker PRISM, which allows us to perform probabilistic yet exhaustive verification.

As a final contribution, we also present a small-scale validation of an architecture of the first category, with end users from three countries (Romania, Poland, Denmark). This work has been carried out with partners from the mentioned countries.

Our work in this thesis paves the way towards the development of user-centered, intelligent ambient assisted living solutions with ensured quality of service.

Sammanfattning

Ambient Assisted Living (AAL) lösningar är riktade för att assistera äldre till ett självständigt och säkert leverne. AAL har under det senaste årtiondet fått ett stort uppsving, mycket tack vare framsteg inom informations- och kommunikationsteknologier, Ubiquitous Computing och Internet of Things (IoT). En närmare granskning av nuvarande AAL lösningar visar dock på att dessa framsteg främst levererar endast en eller ett fåtal funktioner, oftast av samma typ, t.ex. (funktioner för att bevaka hälsa). Det finns jämförelsevis mycket färre initiativ som integrerar olika sorters AAL funktioner som falldetektering, påminnelser, brandlarm etc., förutom hälsobevakning till ett gemensamt ramverk som har intelligent beslutsfattande och därmed bättre förutsättning att kombinera flera olika händelser.

I denna avhandling föreslår vi två olika kategorier av AAL ramverksarkitekturer som implementerar användaranpassade funktionaliteter för att adressera ovanstående problem. Den ena kategorin har en centraliserad approach och använder intelligent Decision Support System (DSS). Den andra kategorin har en distribuerad approach och innefattar flera intelligenta agenter. Den centraliserade arkitekturen är vårt förstahandsval på grund av den enkelheten att utveckla genom att kombinera flera funktionaliteter med ett centraliserat DSS som kan utvärdera beroendes mellan flera händelser I real-tid. Genom att addera ytterligare ett redundant DSS har vi även uteslutit den välkända Single Point of Failure problematiken. Skalbarhet, flexibilitet, självläkande förmåga samt åtkomst för flera användare hos vår centraliserade lösning är svårt att uppnå, vilket är anledningen till att vi även presenterar en distribuerad, agentbaserad arkitektur som andrahandslösning som används vid behov. Båda dessa lösningar kommer att användas i säkerhetskritiska applikationer. Lösningarnas designtidsförsäkran, det vill säga att garantin att de uppfylla kan de funktionella krav som ställs samt leverans av nödvändig servicekvalitet är där-

för fördelaktig.

Vår första lösning är en generisk arkitektur, utformad enligt andra kommersiella AAL-lösningar med sensorer, datasamlare, DSS, säkerhet och integritet, databas (DB) system, användargränssnitt (UI) och Cloud Computing stöd. Vi specificerar Architecture Analysis and Design Language (AADL) via en uppsättning av komponentmönster som vi föreslår. Fördelen med att använda mönster är att de lätt återanvänds när man bygger specifika AAL-arkitekturer. Våra mönster beskriver beteendet hos komponenterna i AADLs beteendeannex och felbeteendet i AADL: s felannex., vi visar även olika instanser av vår generiska modell som kan utvecklas utifrån användarnas krav. Genom att använda hjälp av den toppmoderna modellkontrollen UPPAAL försäkrar vi även att dessa lösningar tillmötesgår de funktionella, tidsmässiga och tillförlitliga kraven. Vi använder även en statistisk modellkontrollsteknik genom UPPAAL SMC vilket är en förlängning av UPPAAL modell checker för stokastiska system som används i de fall då en uttömmande verifiering inte är möjlig. Vår andra arkitektur är en agent-baserad arkitektur för AAL-system, där agenter är intelligenta enheter kommunicerar med varandra för att komma fram till beslut om nödvändiga åtgärder. Beslutsfattandet fördelas nu istället mellan agenter och kan användas av flera användare fördelade på flera platser. Denna lösning kräver dock en beskrivning av agenter samt deras interaktion vilket innebär att den befintliga kärnan AADL inte räcker som enda ramverk. Därför föreslår vi en utvidgning till det centrala AADL-språket - Agent Annexet, som har en formell semantik likt Stochastic Transition Systems, vilket gör att vi kan specificera probabilistiska, icke-deterministiska och real-tids system beteenden inom AAL. Vi använder den toppmoderna probabilistiska modellkontrollen PRISM, som gör det möjligt för oss att utföra en probabilistisk, men uttömmande verifiering av vårt multi-agent system.

Slutligen presenterar vi också en mindre omfattande validering av en arkitektur i den första kategorin, med slutanvändare från tre länder (Rumänien, Polen, Danmark). Detta arbete har utförts med partner från de nämnda länderna.

Vårt arbete i denna avhandling banar väg mot utveckling av användarcentrerade, intelligent ambient-assisted lösningar med garanti för servicekvalitet.

To my husband, Kiran

Acknowledgements

It is with immense gratitude that I write this section. First of all, I would like to sincerely thank my supervisors- Associate Professor Cristina Seceleanu, and Professor Maria Linden for their support, guidance and patience. Thank you for believing in me and giving me an opportunity to undertake PhD studies. Also, special thanks to Dr. Raluca Marinescu for her supervision during her Postdoc employment at MDH. Without all of your guidance and support, this thesis would not have been possible.

Next, I would like to thank all the professors and lecturers at the university for the knowledge they shared. It was a pleasure being with you all and learning new things. Many thanks to my fellow PhD students and the staff at the department. I really enjoyed the time spent with you guys! I would especially like to thank my office mates - Predrag and Nesredin for all the wonderful times we had. I will really miss you guys. Next, I would like to thank the rest of my amazing group - Simin and Rong, you guys are amazing and thanks for all the help and support. I know it is impossible to mention all the people, but this section would be completely meaningless if I don't mention some names. Aida- thanks a lot for all the strong advises and support you have given me. It really means a lot. Gita - thanks for being a wonderful friend. My sincere thanks to Leo, Sveta, Hamidur, Gabriel, Sara, Momo, Filip, Sara A. (2), Jakob, Nabar, Mobyen, Shahina, Elena, Lana etc. etc. for all the helps, friendly chats and discussions.

I would like to thank my opponent Associate Professor Elena Troubitsyna, and the grading committee members Professor Einar Broch Jonsen and Associate Professor Antonio Cicchetti for accepting the invite and taking time in reviewing this thesis.

Last but not least, I would like to thank my family and family friends. To the love of my life, Kiran - words wont suffice to describe what you mean to me

and the support and love you have given me in our 5 years of togetherness. Next to my parents- I cannot express what I feel for them, how much I love them, and how proud I am to be their daughter. To my in-laws, Kiran's mom dad- thanks a lot for accepting me as your daughter, for the freedom and support you have given me. It means a lot. Next, to my brother, Kishore- there is no one like you, thank you for the unconditional love. Heartfelt gratitude to all my school friends, bachelor and master college mates and all the friends we have at India and Sweden for the love and support you have given me. I would also like to thank our family friend, Manoj Bhaskar, for motivating me to apply for PhD positions. And finally, above all, to God, for being my inner strength.

Ashalatha Kunnappilly
Västerås, March, 2019

List of publications

Publications included in the licentiate thesis[1]

Paper A *Do we need an integrated framework for Ambient Assisted Living?*. Ashalatha Kunnappilly, Cristina Seceleanu, Maria Lindén. In Proceedings of the 10th International Conference on Ubiquitous Computing and Ambient Intelligence (UCAmI), LNCS, Springer, pages 52-63, November 2016, Canary Islands, Spain.

Paper B *A Novel Integrated Architecture for Ambient Assisted Living Systems*. Ashalatha Kunnappilly, Alexandru Sorici, Imad Alex Awada, Irina Mocanu, Cristina Seceleanu, Adina Madga Florea. In Proceedings of the 40th IEEE Computer Society International Conference on Computers, Software & Applications (COMPSAC), July 2017, Turin, Italy, IEEE Computer Society, pages 465-472.

Paper C *A Model-Checking-Based Framework For Analyzing Ambient Assisted Living Solutions*. Ashalatha Kunnappilly, Raluca Marinescu, Cristina Seceleanu. MRTC Report, Mälardalen Real-Time Research Center, MDH-MRTC-322/2018-1-SE, March, 2019. *NOTE:* This paper is an extended version of the following article: *Assuring Intelligent Ambient Assisted Living Solutions by Statistical Model Checking*. Ashalatha Kunnappilly, Raluca Marinescu, Cristina Seceleanu. In Proceedings of the 8th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), November 2018, Limassol, Cyprus, Springer, pages 457-476.

[1] The included articles have been reformatted to comply with the licentiate thesis layout.

In Proceedings of the 8th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), November 2018, Limassol, Cyprus, Springer, pages 457-476.

Paper D *Architecture Modelling and Formal Analysis of Intelligent Multi-Agent Systems*. Ashalatha Kunnappilly, Simin Cai, Raluca Marinescu, Cristina Seceleanu. In Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), Crete, Greece, SCITEPRESS, May 2019.

Paper E *An end-user perspective on the CAMI Ambient Assisted Living Framework*. Imad Alex Awada, Oana Cramariuc, Irina Mocanu, Cristina Seceleanu, Ashalatha Kunnappilly, Adina Magda Florea. In Proceedings of the 12th Annual International Technology, Education and Development Conference (INTED), Edulearn, March 2018, Spain.

Additional publications, not included in the licentiate thesis

1. *CAMI - An Integrated Architecture Solution for Improving Quality of Life of the Elderly*. Alexandru Sorici , Imad Alex Awada , Ashalatha Kunnappilly, Irina Mocanu , Oana Cramariuc , Lukasz Malicki , Cristina Seceleanu, Adina Magda Florea. In Proceedings of the 3rd EAI International Conference on IoT Technologies for HealthCare (HealthyIoT), 2016, Springer, LNCS.

2. *Analyzing Ambient Assisted Living Solutions: A Research Perspective*. Ashalatha Kunnappilly, Axel Legay, Tiziana Margaria, Cristina Seceleanu, Bernhard Steffen, Louis-Marie Tranonouez. 12th International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS), 2017, IEEE.

3. *A Formally Assured Intelligent Ecosystem for Enhanced Ambient Assisted Living Support*. Ashalatha Kunnappilly. The 33rd ACM/SIGAPP Symposium On Applied Computing (SAC), Student Research Competition (**Second position**), 2018, ACM.

4. *A Systematic Mapping Study on Real-time Cloud Services*. Jakob Danielsson, Nandinbaatar Tsog, Ashalatha Kunnappilly. IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), 2018, IEEE.

Contents

I

Thesis

Chapter 1

Introduction

According to the statistics of the World Population Ageing Report 2015 [1], the world's elderly population is predicted to reach 2.1 billion by 2050, which is more than double of the population of elderly adults in 2015. The ageing society entails coping with increased health-care costs, shortage of caregivers [2], etc. Ambient Assisted Living (AAL) solutions are gaining popularity in this context, as they can assist the elderly in their daily activities and in their independent living, with limited risks. Some examples of assistance are health monitoring, home monitoring, fall detection, communication with caregivers, mobility, providing recommendations, reminders, etc.

AAL systems are real-time safety-critical systems, i.e., not delivering the right functionality at the right time may have consequences that could even lead to the death of the elderly user. For example, most of the AAL systems use sensors to monitor health parameters like pulse, ECG, blood glucose level, blood pressure, etc. In many cases, health parameter deviations are critical and must be notified to the caregiver in due time, and the failure to do so can endanger the life of the elderly. Hence, early design-stage assurance via techniques like model checking can uncover potential errors before their propagation to implementation levels, or simply provide a guarantee that the design meets the specification.

Upon undertaking a survey of existing AAL solutions [3], we find that many of the existing ones have limited support of functionalities, despite the fact that helping an older adult in his/her daily living requires supporting health-related functions, but also home and social-life related functionalities. Although the above holds, one can use various independent systems providing

one or more of such supporting functions. However, there exist potentially critical scenarios that such solutions cannot resolve in due time, which justify the architectural integration of independent solutions. For instance, if an elderly person's home is equipped with an AAL solution that does not support automatic fall detection, the user can purchase it separately, as there exist readily available solutions that detect a fall and raise an alarm. This functionality of a fall detection system remains the same whether it is an independent system or part of an integrated system. The question that follows is: if the separate solutions can perform their functionality without integration, in isolation, then why would one even think of designing an integrated, more complex solution that in addition comes at a higher price? The most obvious reason for a positive answer would be the increased practicality of a single, integrated solution that offers all the needed support vs. more systems each delivering a particular function, which the users need to purchase. However, there exists a more important reason - the fact that the performance of individualized solutions differ dramatically if they are integrated into a coherent framework, versus the case when they are employed in isolation especially in scenarios where critical events might occur simultaneously. In some cases, independent solutions cannot even depict a potential causality between simultaneous critical events, as we exemplify below.

In our **first** contribution [3], we discuss the behaviors of independent and integrated solutions by selecting representative scenarios that we simulate via sequence diagrams, and check their offline schedules against real-time deadlines. As a result, we conclude that certain critical scenarios can be tackled intelligently only by using **integrated** solutions. For instance, let us assume the following scenarios:

- A fall event occurring due to low pulse: In this case, if the fall sensor and the pulse monitoring sensor work independently of each other, no connection can be established between the two events, indicating that the potential reason for the fall is in fact the person's low pulse, which in turn may be critical for diagnosis.

- Simultaneous occurrence of fire and fall events: When both these events occur together, a safe mitigation of the scenario is achieved only when both these events are communicated to caregivers and firefighters, which is not guaranteed by independent systems working side by side. Assuming that the fire alarm communicated to the firefighters is verified for confirmation by a phone call to the user's home, and since the elderly who has fallen may not be able to answer, the fire alarm may be deemed

false and discarded, triggering a potential catastrophe.

Justified by the above, we establish the fact that the need of integrated AAL solutions that cater for various types of functions is veridical [3]. The next challenge is to develop such systems that can integrate multiple functionalities and deliver them correctly. When AAL solutions are integrated such that they cover a wide variety of functionalities [4], out of which many are safety critical, ensuring the correctness of the system behavior by verifying the functional and quality-of-service (QoS) attributes of the system at the design stage is beneficial. In this thesis, we propose two integrated solutions for AAL systems: a) An architecture with centralized artificial intelligence (AI)-based decision support, and b) An architecture with distributed decision-making using multiple intelligent agents that cooperate with each other. We also show the correctness of the proposed solutions at design level.

The integration of various functionalities can be easily accomplished if there exists a centralized decision maker that all the various devices communicate to, such that different events can be combined in real-time. This prompts us to our **second** contribution as a *centralized* integrated AAL solution that we describe in the Architecture Analysis and Design Language (AADL) [5, 6]. We design our centralized architecture as a generic model that follows the AAL architecture design in the literature by: (i) integrating multiple sensors, data collector unit, decision-support systems, cloud computing facilities, communication gateways and user-interfaces, and (ii) incorporating redundancy to the decision-support systems (local and cloud) to tackle the single point of failure in centralized systems, hence increasing the system's fault tolerance [3, 6]. When modeling this architecture in AADL, we follow a pattern-based modeling approach that facilitates the models' reuse. By using AADL's Behavior Annex (BA), we specify the AI support, combining context modeling, fuzzy logic, case-based reasoning and rule-based reasoning. The combination of the various AI techniques also strengthens the decision-making support of the AAL architecture. Using the AADL model, we perform initial analysis like latency analysis, schedulability, and resource analysis, within the OSATE platform [7].

The generic architecture model can also be customized to address different user requirements and preferences. In this thesis, we show three different instantiated versions of the generic model, that is, a minimal configuration, an intermediate configuration and a complex configuration, modelled in AADL. We give formal semantics to the “semi-formal" AADL modeling constructs of the type used in our work, in the framework of stochastic timed automata [8]. In order to formally analyze the system against various functional and quality-

of-service (QoS) attributes, we show exhaustive verification of the minimal configuration using the UPPAAL model checker and statistical model checking of complex configuration using UPPAAL SMC [9]. The reason for employing statistical model checking is twofold: a) exhaustive model-checking might not scale for large complex systems, and (ii) we model the failure probabilities of various components, and hence the choice of reasoning statistically is justified. Our modeling and verification approach facilitate reuse via a pattern-based modeling infrastructure, covers AI support, and is able to cover a larger set of properties for verification , as compared to existing approaches to AAL system formal modeling and analysis [10, 11]. In addition, most of the commercially available AAL solutions lack a documented proof of correctness [5]. However, our first solution has the same disadvantages as all centralized solutions, that is: (i) redundancy overheads due to ensuring fault tolerance, and (ii) limited scalability and adaptivity.

Our **third** contribution and second architectural solution [12] follows the upcoming trend of using distributed architectures for designing AAL systems, as they provide autonomy, scalability, adaptability and fault-tolerance, in addition to the fact that it servers multiple users at the same time. Hence, we propose a *distributed agent-based AAL solution*, as the second category of architectures that support the design of AAL systems. However, such systems usually possess additional overhead encountered during agent synchronizations for collective decision-making and data consistency maintenance. This overhead can sometimes hamper the real-time behavior of the system. To address this, we investigate how we can use these systems for developing integrated solutions that ensure a safe trade off between autonomous behavior and consistency overheads. This is a challenging requirement since agents are inter-dependent, and have only a limited view of the environment. Concretely, the agent-based solution should ensure a consistent view of the environment, in terms of processed data and events, as well as an inter-agent communication overhead that should not result in breaching the real-time system demands.

Our agent-based architecture consists of independent agents that cater for a particular functionality, respectively, for e.g., a health monitoring agent detects health parameter variations and raise a notification to caregiver. Our architecture supports interactions between different categories of agents. In this thesis, we consider only 2 agent categories: a) simple reflex agents, with reasoning based on if-then-else rules, and b) self-learning intelligent agents, embedded with AI learning algorithms, like Reinforcement Learning [13]. In order for the agents to cooperate in real-time, each agent maintains the dependencies it can have with other agents. For example, if a health-monitoring agent detects

that there is a high pulse, it would need to cooperate with an activity agent to determine the user activity, a high pulse during an exercise session is normal and no notifications should be generated. Hence, the activity agent is included in the dependency list of the health monitoring agent. For formally modeling the agent-based architecture, existing architecture languages such as AADL cannot specify autonomy, adaptability, self-healing, self-learning etc., as these behaviours are usually non-deterministic, probabilistic and have real-time constraints. To describe the agents and the system's architecture, we propose an extension to AADL specification language as a sub-language called *Agent annex*, and define its semantics described in terms of Stochastic Transition Systems [14].

As the **fourth** and final contribution, we also present some initial validation of the centralized architecture by testing some of the implemented functionalities with end-users and in the laboratory [15]. In this contribution, we present an implemented version of the architecture of the first category. The functionalities chosen for implementation are based on user surveys undertaken by the end user organizations within the project. We show the validation results with respect to functionalities like health monitoring (i.e. blood pressure, heart rate, blood glucose, weight, blood oxygenation), fall detection, supervised physical exercises and vocal interactions.

1.1 Thesis Overview

The thesis is divided into two major parts. The first part is an overall summary of the thesis, organized as follows. In Chapter 2, we give a short overview of the preliminaries; in Chapter 3, we describe the research method used for conducting the research and producing the research results described in the thesis. Chapter 4 introduces the research goals of the thesis. In Chapter 5, we briefly describe the contributions of the thesis, and map them to the corresponding research goals, respectively. The overview and comparison to the related work is given in Chapter 6, after which we conclude the first part of the thesis and present the directions for future work in Chapter 7.

The second part of the thesis is given as a collection of publications that encompass all the thesis contributions. The included papers are:

Paper A. *Do we need an integrated framework for Ambient Assisted Living?*. Ashalatha Kunnappilly, Cristina Seceleanu, Maria Lindén. In Proceedings of the 10th International Conference on Ubiquitous Computing and Ambient

Intelligence (UCAmI), LNCS, Springer, pages 52-63, November 2016, Canary Islands, Spain .

Abstract. The significant increase of ageing population calls for solutions that help the elderly to live an independent, healthy and low risk life, but also ensure their social interaction. The improvements in Information and Communication Technologies (ICT) and Ambient Assisted Living (AAL) have resulted in the development of equipment that supports ubiquitous computing, ubiquitous communication and intelligent user interfaces. The smart home technologies, assisted robotics, sensors for health monitoring and e-health solutions are some examples in this category. Despite such growth in these individualized technologies, there are only few solutions that provide integrated AAL frameworks that interconnect all of these technologies. In this paper, we discuss the necessity to opt for an integrated solution in AAL. To support the study we describe real life scenarios that help us justify the need for integrated solutions over individualized ones. Our analysis points to the clear conclusion that an integrated solution for AAL outperforms the individualized ones.

Contributions. I was the main contributor to this work and the main driver for the paper. I performed a literature review of the SOA and SOP of existing AAL solutions and identified that there are very few AAL solutions that are fully integrated w.r.t functionalities chosen based on a multi-national survey conducted in the same research project (CAMI EU project) by end-user organizations. I also performed an analysis of timing requirements for integrated and non-integrated AAL solutions in certain critical scenarios via sequence diagrams and offline schedules. I was helped by the second author to formulate the scenarios and to select the tools for analysis. The second and third authors also provided constructive feedback for the paper.

Paper B. *A Novel Integrated Architecture for Ambient Assisted Living Systems*.Ashalatha Kunnappilly, Alexandru Sorici, Imad Alex Awada, Irina Mocanu, Cristina Seceleanu, Adina Madga Florea. In Proceedings of the 40th IEEE Computer Society International Conference on Computers, Software & Applications (COMPSAC), July 2017, Turin, Italy, IEEE Computer Society, pages 465-472.

Abstract. The increase in life expectancy and the slumping birth rates across the world result in lengthening the average age of the society. There-

fore, we are in need of techniques that will assist the elderly in their daily life, while preventing their social isolation. The recent developments in Ambient Intelligence and Information and Communication Technologies have facilitated a technological revolution in the field of Ambient Assisted Living. At present, there are many technologies on the market that support the independent life of older adults, requiring less assistance from family and caregivers, yet most of them offer isolated services, such as health monitoring, reminders etc; moreover none of current solutions incorporates the integration of various functionalities and user preferences or are formally analyzed for their functionality and quality-of-service attributes, a much needed endeavor in order to ensure safe mitigations of potential critical scenarios. In this paper, we propose a novel architectural solution that integrates necessary functions of an AAL system seamlessly, based on user preferences. To enable the first level of the architecture's analysis, we model our system in Architecture Analysis and Design Language, and carry out its simulation for analyzing the end-to-end data-flow latency, resource budgets and system safety.

Contributions. I was the main driver for the paper. My technical contributions include the AADL analysis of selected architectures from the literature, proposing a novel architecture solution with local and cloud processing as a platform for seamless integration of various AAL functionalities chosen based on Paper 1, evaluation of the proposed architecture in AADL for latency, resource budgets and failure. The other authors contributed with ideas on architecture design, and comments on the paper.

Paper C. *A Model-Checking-Based Framework For Analyzing Ambient Assisted Living Solutions*. Ashalatha Kunnappilly, Raluca Marinescu, Cristina Seceleanu, MRTC Technical Report, Mälardalen Real-Time Research Centre, Mälardalen University, Mälardalen University Press, MDH-MRTC-322/2018-1-SE, March 2019. *NOTE:* This publication is an extended version of the article: *Assuring Intelligent Ambient Assisted Living Solutions by Statistical Model Checking*. Ashalatha Kunnappilly, Raluca Marinescu, Cristina Seceleanu. In Proceedings of the 8th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), November 2018, Limassol, Cyprus, Springer, pages 457-476.

Abstract. Since modern ambient assisted living solutions integrate a multitude of assisted-living functionalities within a common design framework, some are safety-critical, it is desirable that these systems are analyzed

already at their design stage to detect possible errors. To achieve this, one needs suitable architectures that support the seamless design of the integrated assisted-living functions, as well as capabilities for the formal modeling and analysis of the architecture. In this paper, we attempt to address this need, by proposing a generic integrated ambient assisted living system architecture, consisting of sensors, data-collector, local and cloud processing schemes, and an intelligent decision support system, which can be easily extended to suite specific architecture categories. Our solution is customizable, therefore, we show three instantiations of the generic model, as simple, intermediate and complex configuration, respectively, and show how to analyze the first and third categories by model checking. Our approach starts by specifying the architecture, using an architecture description language, in our case, the Architecture Analysis and Design Language that can also account for the probabilistic behavior of such systems. To enable formal analysis, we describe the semantics of the simple and complex categories as stochastic timed automata. The former we model check exhaustively with UPPAAL, whereas for the latter we employ statistical model checking using UPPAAL SMC, the statistical extension of UPPAAL, for scalability reasons. Our work paves the way for the development formally-assured future ambient assisted living solutions.

Contributions. I was the main driver of the paper. My technical contributions include: (i) a generalized architecture framework for AAL systems with centralized decision support, (ii) the pattern-based design of the system architecture in AADL, (iii) the design of an Intelligent Decision Support System combining context modeling, rule-based reasoning, fuzzy logic and case-based reasoning and its pattern-based model, (iii) formal semantics of the AADL patterns, and (iv) verification of the functional specification, and QoS of the system, including its DSS. The other two authors provided ideas w.r.t the modeling and verification of the AAL system, as well as feedback for the paper.

Paper D. *Architecture Modelling and Formal Analysis of Intelligent Multi-Agent Systems*. Ashalatha Kunnappilly, Simin Cai, Raluca Marinescu, Cristina Seceleanu. Accepted in 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2019), Crete, Greece, May 2019.

Abstract. Modern cyber-physical systems usually assume a certain degree of autonomy. Such systems, like Ambient Assisted Living systems

aimed at assisting elderly people in their daily life, often need to perform safety-critical functions, for instance, fall detection, health deviation monitoring, communication to caregivers, etc. In many cases, the system users have distributed locations, as well as different needs that need to be serviced at the same time. These features call for adaptive, scalable and fault-tolerant system design solutions, which are well embodied by multi-agent architectures. Analyzing such complex architectures at design phase, to verify if an abstraction of the system satisfies all the critical requirements is beneficial. In this paper, we start from an agent-based architecture for ambient assisted living systems, inspired from the literature, which we model in the popular Architecture Description and Design Language. Since the latter lacks the ability to specify autonomous agent behaviours, which are often non-deterministic or probabilistic, we extend the architectural language with a sub-language called Agent Annex, which we formally encode as a Stochastic Transition System. This contribution allows us to specify behaviours of agents involved in agent-based architectures of cyber-physical systems, which we show how to exhaustively verify with the state-of-art model checker PRISM. As a final step, we apply our framework on a distributed ambient assisted living system, whose critical requirements we verify with PRISM.

Contributions. I was the main driver for the paper. I designed the intelligent multi-agent system architecture for Ambient Assisted Living Systems (intelligence is incorporated by employing self-learning techniques in agents using Reinforcement Learning). Also, I proposed the extension to the existing AADL modeling framework for modeling agent behaviours- the Agent Annex and formulated its semantics. Further, I also developed the PTA model of the corresponding AADL model that could be model-checked via PRISM. Simin Cai helped in writing the introduction and preliminaries of PRISM and also helped in formatting diagrams, other sections and also gave valuable suggestions and feedback in developing the PTA model and its verification in PRISM. The other authors provided valuable comments and feedback both on the approach and on the final version of the paper.

Paper E. *An end-user perspective on the CAMI Ambient Assisted Living Framework*.mad Alex Awada, Oana Cramariuc, Irina Mocanu, Cristina Seceleanu, Ashalatha Kunnappilly, Adina Magda Florea. In Proceedings of the 12th Annual International Technology, Education and Development Conference (INTED), Edulearn, March 2018, Spain

Abstract. In this paper, we present the outcomes and conclusions obtained by involving seniors from three countries (Denmark, Poland and Romania) in an innovative project funded under the European Ambient Assisted Living (ALL) program. CAMI stands for “Companion with Autonomously Mobile Interface" in “Artificially intelligent ecosystem for self-management and sustainable quality of life in AAL". The CAMI solution enables flexible, scalable and individualised services that support elderly to self-manage their daily life and prolong their involvement in the society (sharing knowledge, continue working, etc). This also allows their informal caregivers (family and friends) to continue working and participating in society while caring for their loved ones. The solution is designed as an innovative architecture that allows for individualized, intelligent self-management which can be tailored to an individual’s preferences and needs. A user-centered approach has ranked health monitoring, computer supervised physical exercises and voice based interaction among the top favoured CAMI functionalities. Respondents from three countries (Poland, Romania and Denmark) participated in a multinational survey and a conjoint analysis study.

Contributions. The second author was the main driver of the work. I contributed to the paper by proposing a smaller implementable version of the CAMI architecture proposed in Paper 2, took part in DSS implementation and in testing the fall detection functionality using Vibby sensors in laboratory. I have also contributed to the writing and reviewing of the paper, along with the other co-authors.

Chapter 2

Preliminaries

In this chapter, we introduce the preliminary concepts that are used throughout the thesis. First, in Section 2.1 we present the Architecture Analysis and Design Language. Next, in Section 2.2 we give an overview of agents and multi-agent systems. In Section 2.3, we present an overview of the formal modeling, verification and analysis techniques and tools used in the thesis.

2.1 Architecture Analysis and Design Language

AADL [16] is a textual and graphical language in which one can model and analyze a real-time system's hardware and software architecture as hierarchies of components at various levels of abstraction. There are three categories of component abstractions in AADL: Application Software (*Process, Data, Subprogram, Thread,* and *Thread Group*, etc.), Execution Platform (*Device, Bus, Processor, Memory,* etc.), and general composite components (*System* and *Abstract*). *System* components are the top-level components. A *process* component contains a set of *thread* components that define the dynamic behavior of the process. AADL component categories like *Application Software*, *Execution Platform* and *System* are used to represent the run-time architecture of the system, however a more generalized representation is possible by specifying it as *abstract*.

A component in AADL can be defined by its *type* and *implementation*: the component type declaration defines the interface of the component and its externally observable attributes, whereas the component implementation defines

its internal structure. AADL allows possible component interactions via *ports/features*, *shared data*, *subprograms*, and *parameter connections*. In AADL, the input/output ports can be defined as: *event ports*, *data ports*, and *event-data ports*. Based on the component interactions, explicit *control flows* and *data flows* can be defined across the interfaces of AADL components by specifying the components as *flow source*, *flow path* or *flow sink*. The components can also be associated with various *properties*, like the *period* and *execution time* and the *dispatch protocol*. The *dispatch protocol* specifies if the component trigger is *periodic* or *aperiodic*.

The functional and error behavior of a component are described by the *Behavior Annex (BA)* [17] and the *Error Annex (EA)* [18] respectively, which model behaviors as transition systems. The BA state machine interacts with the component interface and represents the system behavior. Given finite sets of states and state variables, the behavior of a component is defined by a set of state transitions of the form $s \xrightarrow{guard,\ actions} s'$, where s, s' are *states*, *guard* is a boolean condition on the values of state variables or presence of events/data in the component's input ports, and *actions* are performed over the transition and may update state variables, or generate new outputs. Similarly, the EA models the error behavior of a component as transitions between states triggered by error events. It is also possible to represent the different types of errors, recovery paradigms, probability distribution associated with the error states and events, and also specify error flows and propagations within the component, and between various components.

An *abstract* component allow us to defer from the run-time architecture of the system. The need for this generic model stems from the fact that in real-world applications like AAL, it is difficult to assign run-time semantics to components before the design matures. These generic component categories can be parameterized, and can be refined later in the design process through the "extends" capability of AADL. AADL allows us to archive these components and reuse them. For this, we partition them into two public packages in AADL, namely *component library* and *reference architecture* [19]. A *component library* creates a repository of component types and implementations with simple hierarchy. It can be established via two packages: (i) *Interfaces Library* comprising generic components like sensors, actuators and user-interfaces (UI), and (ii) *Controller Library* that includes the control logic. The *Reference architecture* creates a repository of components of complex hierarchy, e.g. the top-level system architecture.

The AADL core language is designed to be extensible and can be extended via *user-defined properties* and *annex sub-languages*. *User-defined properties*

are relatively simpler extensions, when compared to sub-languages, and can be associated with modeling elements as simple values, for instance, integers or strings. However, *sub-languages* allow more complex structures to be added to an AADL model. A sub-language can be standardized and published as an *AADL annex*. Several such annexes have been defined, for example, the *behavior annex* to model the component's behaviour, and the *error annex* for modeling the error behaviour of the system. *Annex sub-languages* are included into AADL specifications as *annex libraries* or *annex sub-clauses*. An annex library is used to define classifiers defined in an anonymous namespace, or in a public or private part of a package. *Annex sub-clauses* are inserted into component types and component implementations and can reference the classifiers declared in the annex library. In AADL, annexes are considered to be separate from the core AADL, i.e., if we remove all the annex libraries, sub-clauses, and annex-related property associations, the resulting model is a valid core AADL model. Moreover, the different annexes are assumed to be independent of each other.

2.2 Multi-Agent Systems

Although we can define agents from different perspectives, in this thesis we define agents as entities that can sense the environment via sensors and act on the environment via actuators. As described in the literature [20] [21], an agent can be characterized by the following properties:

- **Autonomy**: Agents are autonomous entities, that is, they are capable of taking an independent action without human intervention to meet their respective functionalities.
- **Cooperation:** Cooperation is an inherent property of agents, where each agent has only a limited view of the environment and needs to cooperate with the other agents by exchanging information in order to make a decision. In our work, we consider agent cooperation via a dependency relation that each agent has with other agents and we refer to it by the term *agent dependencies*. Agent dependencies can be dynamic or static, i.e, they may or may not change in time. We assume that the agents establish the cooperation (communication) via message-passing.
- **Responsiveness:** By responsiveness, we mean that the agents should be able to perceive the changes in the environment over time and take timely actions accordingly.

- **Learning:** This is the property of some intelligent agents who learn over the course of their interactions with the environment and thereby produce an increased performance over time. Specific learning algorithms like supervised learning, unsupervised learning, or reinforcement learning can be utilized for imbibing learning into the agents. In this paper, we consider rule-based reasoning algorithms (which are simple if-then-else rules) and reinforcement learning [22] techniques.

- **Adaptivity:** Agent systems should be adaptive, i.e., even if an agent fails, the other agents should be able to carry out their respective functionalities. In our architecture, this is employed by deploying multiple redundant agents such that if an agent fails, another one can take over.

- **Mobility:** Some agents (e.g., robots) are capable of moving from one location to another and we indicate this via the mobility property. At the moment, we consider only stationary agents.

Among the different types of agents available, we consider only Reflex agents and Learning agents. The salient features of these two agent types are discussed briefly below.

1. **Reflex agents:** Reflex agents are based on condition-action rules to decide on the action it should enforce on the environment based on sensor data. They are the fastest, but fail completely if the environment is not fully observable.

2. **Learning agents:** These agents have mechanisms to improve their actions over time. They monitor their actions each time via a performance element and at any point of time, suggests an action that will improve the system performance. In our work, we incorporate learning via reinforcement learning.

A network of agents is referred to as a *Multi-Agent System (MAS).* In a MAS, different types of agents interact with each other to achieve a common or a conflicting goal [23]. The interaction between the agents can be direct or indirect. In direct communication, agents can send messages directly to each other and are responsible for their own coordination, however communication cost and implementation complexity (in case of large MAS) are the major disadvantages of direct communication. An alternative approach is to use mediator for communication between the agents (indirect communication). In this thesis, we use

a combination of direct and indirect communication to achieve the agent interactions in our MAS. The mediator (in our case, *a tracker*) is responsible for maintaining agent locations and addresses and thereby establishing the MAS coordination. Once an agent establishes a coordination to another agent via the tracker to achieve a goal (via dependency relation), the further communications are handled directly. In case the tracker fails, the direct communication between the agents are established for achieving the necessary coordination and data consistency.

2.3 Formal Modeling and Verification by Model Checking

Formal Modeling and Verification relies on a set of mathematical techniques that are used to rigorously prove the correctness of a system model expressed in some formal notation. Formal verification techniques are deemed to deliver a higher degree of assurance when compared to other verification techniques such as simulation and testing.

One of the most popular formal verification techniques, *model checking*, an automated technique that checks a finite-state abstract system model in a systematic and exhaustive manner, to prove whether it satisfies a given property modeled in logic. Model checking if fully automated and is performed by a verifier tool called *model checker*. The core of model checking is the verification algorithm, performed by the model checker. The input to the model checker is a system model expressed in a formal notation and a set of formally specified logical properties. For verification of qualitative properties (that admit a yes/no answer) there are two possible outcomes of the model checking procedure. If the model conforms to a given property, the model checker returns a positive answer. For reachability and some liveness properties (e.g., something good will eventually happen) the model checker returns a witness trace in case of fulfillment. Then, the model checking activity can be continued for the rest of system properties. When a safety property is not satisfied, the model checker generates a counter example, which is usually a path (error trace) to the state that violates the property.

Due to its systematic approach and the exhaustiveness of the state space exploration, the model checking procedure can handle models with state spaces up to a certain size, above which there is not enough memory to store new states. This is known as the state space explosion problem.

In this thesis, we apply model checking on architectures of ambient as-

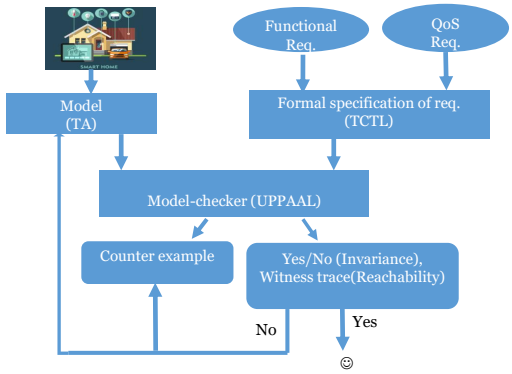


Figure 2.1: Model checking procedure.

sisted living system that we propose, and show our approach on minimal as well as more complex configurations. For minimal architectures, the exhaustive model-checking method scales. The exhaustive model checking, employing the model checker UPPAAL, is shown in Figure 2.1. For complex architectures that integrate multiple AAL functionalities and have several component connections and users, the exhaustive model checking is likely not to scale. Therefore, we resort to a special type of model checking called *statistical model checking* (SMC), which offers the guarantee that a model satisfies a given property up to some probability, based on a finite number of model simulations. A high-level overview of the SMC, usally employed by statistical model checkers like UPPAAL SMC, is given in Figure 2.2. SMC uses a series of simulation-based techniques to answer two types of questions: i) *Qualitative*: is the probability of a given property being satisfied by random system executions greater or equal than some threshold? and ii) *Quantitative:* what is the probability that a random system execution satisfies a given property? The qualitative properties are also referred to as *hypothesis testing*, while the quantitative are called *probability estimation*. In both cases, the answer provided by the procedure will be correct up to a certain level of confidence. Since statistical model checking is less memory intensive than traditional model checking, it can be used to statistically verify models with infinite state spaces. Even though the technique is less precise than the exact model checking, it still solves the verification problem in a rigorous and efficient way.

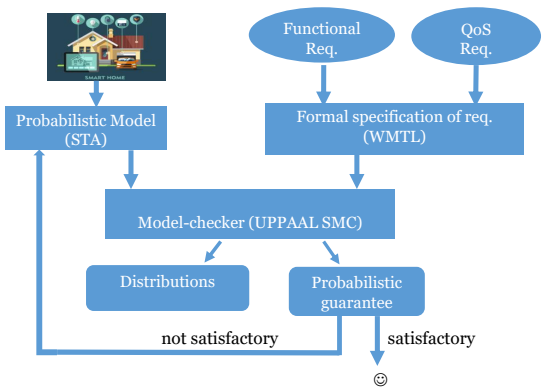


Figure 2.2: Statistical model checking procedure.

In our work, we use the model checkers - UPPAAL [24] and PRISM [25] for exhaustive model-checking and UPPAAL Statistical Model Checker (SMC) [8] for statistical model-checking of complex models. In the following subsections, we briefly overview the timed automata, probabilistic timed automata and stochastic timed automata frameworks, and the temporal logics used for specification of the system properties in the respective model checkers.

2.3.1 Formal Modeling Frameworks

Timed Automata. Timed automata (TA) [26] formalism is an extension of finite-state automata with a set or real-valued variables called *clocks*, suitable for modeling the behavior of real-time systems. The clocks are non-negative variables that grow at a fixed rate with the passage of time, and can be reset to zero.

The semantics of TA is defined as a *timed transition system* $(S, \rightarrow)$, where S is a set of states and $\rightarrow$ is a transition relation that defines how the system evolves from one state to another. A state in the system is a pair (l, v), where l is the location and the v is the valuation of the clocks. A timed automaton can proceed, that is, move to a new state, by performing either a *discrete* or a

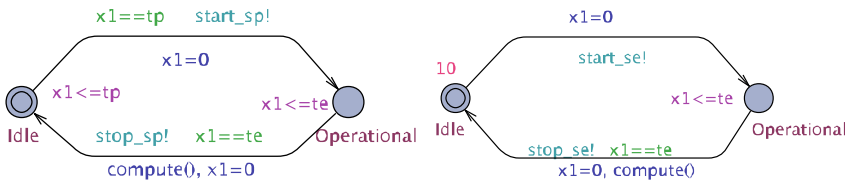
delay transition. By executing a *discrete* transition the automaton transitions from one location into another without any time delay, whereas by executing a *delay* transition the automaton stays in the same location while time passes.

A system can be modeled as a set of communicating components. Let $A_1, A_2, ...A_n$ be a set of timed automata each corresponding to an individual component in the system. A *network* of timed automata (NTA) is simply a parallel composition $A_1 \| A_2 \| \cdots \| A_n$ of a finite number of timed automata.

Next, we present the timed automata variants used by the UPPAAL model checker, UPPAAL SMC model checker, and PRISM model checker by means of examples.

UPPAAL Timed Automata. UPPAAL TA [24] extends TA with discrete variables as well as other modeling features, like urgent and committed locations, synchronization channels, etc. A real-time system can be modeled as a network of TA composed via the parallel composition operator ($\|$), which allows an individual automaton to carry out internal actions, while pairs of automata can perform handshake synchronization. UPPAAL model checker [24] provides exhaustive model checking for UPPAAL TA. The formal definitions of its syntax and semantics can be found in our Paper C [6].

The automaton in Figure 2.3a shows an example of an ordinary UPPAAL TA that models the behavior of a periodic sensor executing some computational routine (`compute()`) that maps inputs into outputs. It has two locations: `Idle` and `Operational`, out of which `Idle` is marked to be the initial one, denoted by two concentric circles. `Idle` is decorated with an invariant `x1` $\leq$ `tp`, denoting that the automaton is allowed to stay in that location as long as the value of the clock variable (`x1`) is smaller or equal to the value of the period (`tp`). The edge from `Idle` to the `Operational` location is decorated with the guard (`x1==tp`). It also has an `update` action, in this particular case being a reset of the clock variable `x1` and a `synchronization action` (`start_sp!`) to synchronize the start of the sensor with the rest of the system. The `Operational` location is decorated with an invariant `x1` $\leq$ `te`, denoting that the automaton is allowed to stay in that location as long as the value of the clock variable is smaller or equal to the value of the execution time (`te`). The `Operational` location represents the operational mode of the automaton and has a transition decorated with a *guard* expression `x1 == te`. On the same edge two update actions are performed, namely executing the computational routine that produces output from the execution (`compute()`), and reset of the clock variable. The computational routine is encoded as a `C` function.



(a) Timed Automata (b) Stochastic Timed Automata

Figure 2.3: Illustrative scenario of UPPAAL TA and UPPAAL SMC TA

UPPAAL SMC Timed Automata. UPPAAL SMC TA [9], referred to as stochastic timed automata (STA) is a formalism defined as the stochastic interpretation of the TA and refines it with: (i) probabilistic choices between multiple enabled transition, where the output *probability* function γ may be defined by the user, and (ii) probability distributions for non-deterministic time delays, where the *delay density function* μ is a uniform distribution for time-bounded delays or an exponential distribution with user-defined rates for cases of unbounded delays.

UPPAAL Statistical Model Checker (UPPAAL SMC) [9] provides statistical model checking for STA. A model in UPPAAL SMC consists of a network of interacting STA (NSTA) that communicate via broadcast channels and shared variables. In the network, the automata repeatedly race against each other, that is, they independently and stochastically decide how much to delay before delivering the output, and what output to broadcast at that moment, with the "winner" being the component that chooses the minimum delay.

Figure 2.3b shows an example of a timed automaton with stochastic semantics, in our case, a sensor that operates aperiodically. The automaton is composed of the same two locations (`Idle` - the initial one, and `Operational`) like the example we had in Figure 2.3a. To model the aperiodic behavior of the component, instead of an invariant, the `Idle` location is decorated with a *rate of exponential*. The distribution parameter λ is the user-defined parameter in the delay function that calculates the probability of the automaton leaving the `Idle` location at each simulation step, given as: $Pr(\text{leaving after } t) = 1 - e^{-\lambda t}$. The greater the value of λ, the smaller is the probability that the automaton stays in the location.

In this thesis, we use NSTA formalism to model our centralized AAL architecture exhibiting random failures.

PRISM Probabilistic Timed Automata. Probabilistic Timed Automata (PTA) as modelled by PRISM model checker [25] supports formalism to model systems that exhibit probabilistic, non-deterministic and real-time characteristics.

Listing 2.1 represents an excerpt of the PTA model of a periodic sensor. The variable `s1` represents the location, `s1=0` represent the initial location `Idle`, and `s1=1` represent the location `Operational` and `s1=2` represent the `Failed` location indicating the sensor failure. `x1` is the `Clock` variable. The `invariant` in location s1=0 represent the periodic activation of the sensor and that in location s1=1 represent the sensor execution time. The `transitions` of the system indicate that the sensor gets periodically activated and it has a probability 0.999 to go to `Operational state (s1=1)` and 0.111 probability to go to `Failed state (s1=2)`

Listing 2.1: PTA Model in PRSIM of a periodic sensor

```
pta
module sensor
  s1: [0..2] init 0; //states 0-Idle,1-Operational,2-Fail
  x1: clock;
  invariant
   (s1=1 => x1<=2) & (s1=0 => x1<=1)
  endinvariant
  [1]s1 =0 & x1=1-> 0.999:(s1'=1)
    & (x1'=0) + 0.001:(s1'=2) &(x1'=0)
endmodule
```

A *system* is defined as a network of modules via parallel composition: $Sys = PTA_1||...||PTA_n$. A global state is the valuation of all variables of all modules. A module can both read and write its own local variables, but only has read access to the local variables of other modules. Synchronized transitions of modules are identified by the commands with the same labels.

In this thesis, we use PTA models to represent our AAL multi-agent system.

2.3.2 Model-checking Tools

UPPAAL

UPPAAL [24] is an integrated development environment for modeling, simulation and verification of real-time systems. It has been developed as a joint research effort by the Uppsala University and Aalborg University. The tool has been first released in 1995 and since has been constantly updated with new features. The properties to be verified by model checking the resulting network of timed automata are specified in a decidable subset of (Timed)

Computation Tree Logic ((T)CTL) [27], and checked by the UPPAAL model checker. UPPAAL supports verification of liveness and safety properties [24]. The ((T)CTL) queries that we verify in this thesis are of the form: i) **Reachability**: $E\Diamond p$ means that there exists a path where p is satisfied by at least one state of the path, and (ii) **Time bounded Leads to**: $p \rightsquigarrow_{\leq t} q$, which means that whenever p holds, q must hold within at most t time units thereafter.

UPPAAL SMC

UPPAAL SMC [9] is an extension of UPPAAL tool that supports the model-checking of timed-automata networks with stochastic semantics. Unlike the exhaustive model-checking performed by UPPAAL, SMC performs statistical model-checking. The SMC algorithms are less memory intensive, and do not suffer from the state space explosion problem. UPPAAL SMC uses an extension of weighted metric temporal logic (WMTL) [28] to provide probability evaluation $Pr(*_{x \leq C} \phi)$, where $*$ stands for $\Diamond(eventually)$ or $\Box(always)$, which calculates the probability that ϕ is satisfied within cost $x \leq C$, but also hypothesis testing and probability comparison.

PRISM

PRISM [25] is a probabilistic model checker that allows model-checking of systems with random or probabilistic behaviour. Although PRSIM supports model checking of various categories of probablistic models like discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs), Markov decision processes (MDPs), probabilistic automata (PAs) and probabilistic timed automata (PTAs), we employ it only for the PTA models discussed in the previous section. In PRISM, a PTA is represented by a *module*. The property specification language of PRISM for PTA is based on Probabilistic Computation Tree Logic (PCTL) [29]. The model checker can verify whether the probability of a path property pp is within a bound b, which is specified as: $P\, b\, [\, pp\,]$. Here, b can be any of $>= p$, $> p$, $<= p$ or $< p$, where p is a double within [0,1]. A path property pp is a formula that evaluates to either true or false for a single path in the model, in which one can apply the following operators: X (next), U (until), F (eventually), G (always), W (weak until), R (release). PRISM can also compute the minimum and maximum probabilities of a path property, in the form of: $Pmin =?\, [\, pp\,]$, and $Pmax =?\, [\, pp\,]$, respectively.

Unlike the statistical model-checking employed in UPPAAL SMC, which yields approximate results, PRISM supports exhaustive model-checking of

probabilistic models and hence can yield concrete results. However, it suffers state space explosion problem while model-checking of complex models. Moreover, PRISM does not have a simulator implemented for PTA models which makes the debugging of complex models extremely difficult. Finally, it should be remarked that in contrast to UPPAAL, PRISM does not have a graphical representation of its models, which are completely defined in a textual way.

Chapter 3

Research Methodology

In this chapter, we present the research methodology that describes the various steps followed to address our research goal. In Computer Science, a research process is described by four iterative steps: (i) formulating the research problem, (ii) proposing the solution, (iii) implementing the solution, and (iv) validation [30]. Solving a research problem is an iterative process, allowing feedbacks between stages.

The overview of the research methodology used in this thesis is described in Figure 3.1. As shown, the main steps are as follows: (i) identifying the research problem, (ii) formulating the overall research goal, (iii) formulating research questions that address the goal, (iv) proposing and implementing solutions to tackle the research questions, and (v) validating the proposed solutions on relevant use cases and users.

As a first step, we identify the research problem. This is done by performing an extensive literature survey [31] to identify the state-of-art (SOA) and the state-of- practice (SOP) in the area of research. The literature survey involves identifying and reading potential publications in the form of journals, conferences, workshops, PhD dissertations, peer reviewed reports related to the domain. In this thesis, we have conducted an extensive literature survey by the so-called *critical analysis of literature* method [32], in the field of AAL. The literature study has also involved analysis of certain user scenarios. With the study, we have identified the potential research problems in the field of AAL, the major ones being the lack of fully integrated systems and rudimentary AI decision support of existing systems, limited user involvement, and lack of formal assurance of the existing systems. This has helped us to formulate

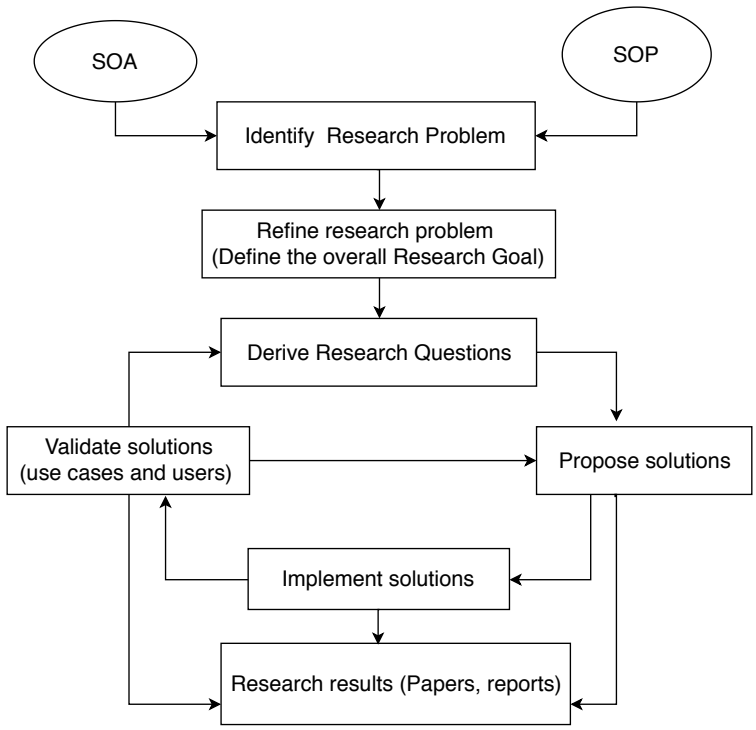


Figure 3.1: Our research process.

our research goal, focusing on the development of integrated AAL solutions with enhanced AI reasoning, sufficient user involvement and formal assurance. Once the research goal has been established, to narrow down the focus, we have identified a set of tailored research questions that need to be answered in order to meet our overall goal. The research questions are presented in Chapter 4.2 . After this step, we have moved to addressing the smaller research questions by developing solutions, presenting the achieved research results and comparing these research results with the research questions. In developing our solutions we have drawn ideas from the related work. Our research results include proposing architecture frameworks for intelligent AAL systems and their formal specification and verification. In this thesis, we use a combination of various research methods, i.e., *proof of concept*, *inductive method*, *model-building* and *simulations of models*, *analysis*, *examples* [32, 33, 34]. For

formulating the AAL architectures, we have used the *proof of concept* research method. While modeling the AAL architectures, we have used *model building* and *simulation of models* approaches.

The last stage of our research process is validation. For formally verifying the properties of AAL architectures, we have used the *analysis method* of validation. However for analyzing various case-studies to evaluate the effectiveness of existing solutions, we have used the *inductive research method*. We have also used *case studies* to validate our proposed architecture with end-users.

Chapter 4

Research Problem

In this chapter, we define the research problem, the overall goal and the research questions of the thesis. In Chapter 4.1 we describe the research problem, after which in Chapter 4.2 we define the overall research goal based on the actual state of practice and state of the art. To narrow the over-arching goal, we define in the same section research questions that help us to structure our research and relate the results to the problem.

4.1 Problem Definition

The world population is rapidly increasing across the world [1]. This demographic trend is followed by new challenges in the society, like increasing number of diseases, increased health-care costs, shortage of caregivers, etc. Besides, it is also witnessed that almost 89% of elderly adults like to live within the comfort of their own homes [2]. Such facts have motivated the research community to focus on the so-called "Ambient Assisted Living" paradigm, which aims to develop intelligent assisting solutions that help the elderly in their safe and independent living, while ensuring that they are not socially isolated.

We have carried out a survey of the state-of-the-art and state-of-practice with respect to AAL solutions, which results in the clear conclusion that there exists a potential research gap in the design and development of a user-tested solution for AAL that integrates various relevant functionalities (e.g. health monitoring, smart home, reminders, fall detection, telepresence etc.) but also

caters for the possible critical situations in a timely manner [3]. Assuming that particular critical events might occur simultaneously, we have analyzed the behaviors of existing individualized or partially-integrated solutions working side by side and concluded that they are not able to tackle particular critical, concurrent events, in real time. One example of a critical scenario is the occurrence of fire and fall events simultaneously, which according to our study cannot be safely resolved by employing independent systems. In such scenarios, a safe resolution can be achieved only when the occurrences of both events are communicated to both caregivers and firefighters, who can then further communicate and prioritize their actions accordingly. If the firefighters are informed only of the fire event and not of the person's fall too, and assuming that answering a telephone call is the way to confirm that the event is veridical, they might deem the fire alarm false and decide not to take action, which can possibly result in loss of life.

Moreover, most of the existing AAL solutions are not necessarily backed by user-acceptance studies, and there is no formal-analysis-based evidence of their functional and timing correctness, which given the connected and distributed nature of most AAL systems is no trivial job. Given the above motivation and challenges, our overall thesis goal is to propose intelligent, integrated and user-centered AAL solutions with ensured functional and extra-functional requirements.

4.2 Research Goals

Based on the above discussed problems, we formulate the overall research goal of the thesis as follows:

Overall Research Goal. *Facilitate the integrated support for achieving self-management of the elderly people by using intelligent ambient assisted living solutions with ensured quality-of-service.*

The overall goal aims to develop integrated solutions for enhancing the support given to elderly adults living independently in their homes and provide a formal assurance to such assisted living frameworks at the design stage of development. Nevertheless, it is obvious that the overall goal is highly abstract and broad. To narrow down the goal and to be able to measure the contributions, we divide it into three research questions in the following.

In order to gain insight into the existing AAL systems and discover their potential challenges, we formulate the first research question as follows:

Research Question 1. *What are the main characteristics, strengths and limitations of existing AAL solutions?*

To answer this research question, we conduct a literature survey of existing AAL solutions and establish a need for an integrated, flexible, and user-centric AAL solution that should include health monitoring, home management, as well as communication and socialization. This leads to our second research question that focuses on how to design such a solution with ensured QoS. This research question is divided into two sub-questions. Research Question 2a focuses on designing intelligent AAL solutions that integrate various health and home management functions. Once the AAL solutions are designed, it is crucial to ensure their QoS and timely response, hence we formulate the Research Question 2b on how to guarantee the function and the considered QoS of the proposed solutions.

Research Question 2. *How can we achieve an intelligent AAL architecture with ensured behavior, timeliness and reliability, which integrates various health and home management functions?*

Research Question 2a. *How do we design architecture frameworks that allow for the integration of multiple functionalities to achieve intelligent decision making in real time?*

Research Question 2b. *How can we employ formal specification and verification technologies to provide certain level of assurance to the integrated solutions with respect to functional correctness and other QoS attributes?*

The outcome of addressing Research Question 2 should be an approach or framework for developing formally assured integrated AAL systems, starting from the architecture design, their specifications and formal verification that can provide guarantees for the critical functional and quality-of-service requirements at early stages of development.

As a final step, we need to validate our proposed solutions on real-life scenarios with representative users. Thus we formulate our Research Question 3 as follows.

Research Question 3. *How do we validate the suitability of the proposed AAL solutions with respect to various AAL scenarios?*

The outcome of addressing Research Question 3 should be the validation or testing of the proposed AAL system in laboratory settings and with real-users and collecting their feedback.

Chapter 5

Thesis Contributions

In this chapter, we give an overview of the research results and contributions that address the research questions defined in Chapter 4.2. The main contributions of the thesis are on four fronts: i) a literature survey of the existing AAL solutions to identify their pros and cons; ii) an integrated architecture design for AAL systems with a centralized decision support, and a formal analysis framework for such systems, based on model checking the architectural specifications and behavior, using UPPAAL and UPPPAL SMC; iii) an agent-based architecture for AAL systems and a method for its formal analysis using PRISM, and iv) a small-scale validation of the proposed architecture of the proposed centralized solution with end users.

5.1 Literature Survey of Existing AAL Solutions

In our first contribution, we address the motivation and background of our study, and thereby tackle RQ 1. The literature study that we undertake compares some AAL solutions against the required functionalities of an AAL system. The functionalities for analysis are chosen based on a multi-national survey with participants from Poland, Denmark and Romania, carried out in the EU project CAMI [1]. The AAL functions that we consider are as follows: (1) health monitoring, (2) fall detection, (3) communication and socialization, (4) support for supervised physical exercises, (5) personalized intelligent and dynamic program management, (6) robotics platform support, (7) intelligent

[1] http://www.camiproject.eu/

Table 5.1 Functionalities supported by various AAL frameworks

AAL Platforms	1	2	3	4	5	6	7	8	9	10
inCASA	✓	✓	✓	✗	✗	✗	✓	✗	✗	✓
iCarer	✗	✗	✓	✗	✓	✗	✓	✗	✗	✓
Persona	✗	✗	✓	✗	✓	✗	✓	✓	✗	✓
Reaction	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗
UniversAAL	✓	✓	✓	✗	✓	✗	✓	✗	✗	✗
iDorm	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓
Robocare	✗	✓	✗	✗	✓	✓	✓	✓	✗	✓
Aware Home	✗	✗	✓	✗	✓	✗	✗	✗	✗	✓
Mav Home	✗	✗	✗	✗	✓	✗	✓	✗	✗	✓
CASAS	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓
GiraffPlus	✓	✓	✓	✗	✓	✓	✓	✗	✗	✓
MobiServ	✗	✓	✓	✓	✓	✓	✗	✓	✗	✓

personal assistant that takes orders, gives advice and reminders, (8) support for vocal interface, (9) mobility assistance, and (10) home and environment management. Table 5.1 shows the functionalities supported by some of the existing AAL frameworks. As shown, none of the existing frameworks support a complete integration of all the functionalities. To establish the need for an integrated architecture for AAL systems, we analyze the timing requirements of integrated and non-integrated AAL solutions in certain critical scenarios, involving simultaneous occurrence of fire and fall events. The analysis is carried out via sequence diagram simulations in Visual Paradigm [35] and by computing their offline schedules. At the end of the analysis, we conclude that there are potential critical scenarios, that can only be tackled by fully integrated AAL solutions.

As a first step targeted towards the design of integrated architectures in AAL, we present a feature diagram representation [36] of functions of an integrated AAL system. We also analyze the existing architecture frameworks that support integration of multiple functionalities [37, 38]. Our analysis considers two types of existing architecture solutions: one with a centralized decision maker [37] and the other with distributed decision making entities, i.e., agents [38]. We employ the Architecture Analysis and Design Language (AADL) to model both architectural variants, and the AADL's analysis capabilities to compare the latencies of the selected frameworks. Our analysis points to the

conclusion that centralized architectures are straightforward solutions to integrate multiple AAL functionalities in real time (due to the lower communication and consistency overheads accounted during the latency analysis, when compared to distributed architectures). Nevertheless, the agent-based solution with distributed decision making bears the advantage of better scalability and adaptivity.

These contributions are addressed in Paper A [3] and Paper B [5], and answer to RQ 1.

5.2 A Centralized Integrated Architecture for Ambient Assisted Living and a Framework for its Formal Assurance

The second contribution of our work is a novel integrated architectural framework for AAL, with a centralized decision support system (DSS), and its formal assurance framework.

Centralized Architecture for integrated AAL Functions. We develop the centralized architecture as a generic model that can be easily instantiated to suit different system requirements. The generic architecture model is inspired by commercial AAL systems with various sensors for home and health monitoring, a data collector, DSS, security and privacy, database (DB) systems, user interfaces (UI), and cloud computing support. The architecture is presented in Fig. 5.1 and supports four different sensor categories: a) Sensor_A: Wearable sensors that send information as data (W_data), e.g., sensors measuring health parameters like pulse, ECG, etc.; b) Sensor_B: Non-wearable sensors measuring ambient parameters and health parameters (NW_data), e.g., camera sensors, motion sensors, etc.; c) Sensor_C: Wearable sensors that detect events (W_event), e.g., fall sensors; d) Sensor_D: Non-wearable sensors detecting events (NW_event), e.g., fire sensors. The data from the sensors are collected by the Data Collector unit. The latter does some data processing and assigns labels and priorities to the incoming data. The Data Collector sends the data to the message queue in the Local Controller, where it gets sorted according to its priority such that when the DSS processes the first element in the queue, it processes the message with the highest priority. Our architecture has both local and cloud-based processing in order to ensure fault tolerance with respect to DSS. The components of the architecture can interact via various commu-

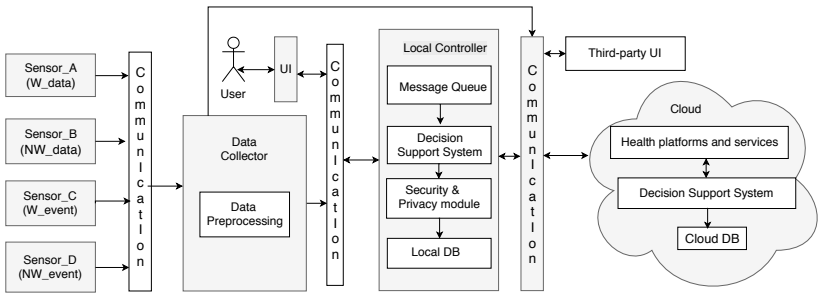


Figure 5.1: A centralized integrated architecture for AAL

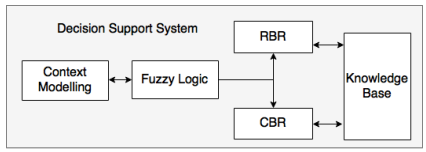


Figure 5.2: An intelligent context-aware Decision Support for AAL Systems

nication protocols. Two distinct features of the architecture, as compared to other AAL architectures, are: (i) the presence of both local and cloud-based processing schemes, and (ii) the continuity of services, even in the absence of Internet (due to the local processing of data).

The core of our system is the **intelligent context-aware DSS**, shown in Fig.5.2. The novelty of our DSS stems from the combination of various AI algorithms, like rule-based reasoning (RBR), fuzzy logic, and case-based reasoning(CBR) with context reasoning for efficient decision-making. The context-reasoning module of the DSS is capable to react to the rapidly changing contexts. We use fuzzy reasoning to identify deviations in health parameters. For instance, the normal pulse range of a person is between 60-100 heart beats per minute. If a person has a pulse measure of 59.5 or 100.5, it should be still considered normal and should not raise any pulse deviation alarms. However, the normal boolean logic, cannot handle the scenario, which can only calssify 59.5 and 120.5 as abnormal pulse values. Hence, we employ fuzzy reasoning with RBR, where a crisp classification can be avoided. With fuzzy reasoning, we can provide degree of memberships, i.e, a pulse value of 59.5 can be considered 97% normal and 3 % abnormal, making the reasoning more effective. The inference engine is a combination of RBR and CBR. RBR is based on

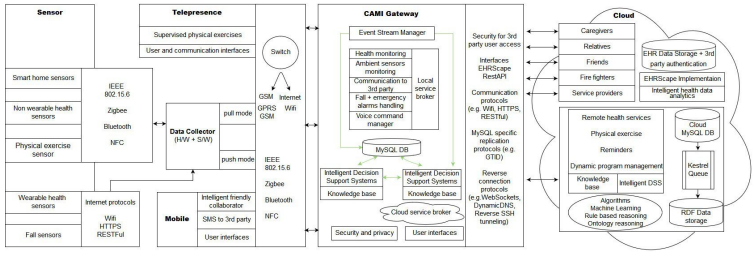


Figure 5.3: The CAMI architecture

simple *if-then-else* rules, which works effectively in cases of well-defined domain knowledge. For instance, " if a fire is detected, then alert the firefighter". However, RBR systems fails completely if there is no specific rule to handle a scenario and in situations where we need to learn and adapt. Such a situation would be, for instance, making a personalized medical recommendation for a user, which may vary from person to person. In this case, CBR reasoning, which can adapt and learn from previously-solved cases can be better suited. In our DSS, we allow RBR to handle the context first, followed by the CBR. The case base of the CBR system is built with successful RBR outputs.

In this thesis, we show three different instantiated configurations of the generic architecture model and their DSS as follows:

- A minimal configuration that uses two sensors - a pulse monitoring sensor wearable and fall sensor wearable, a data collector, a mobile phone UI, and a cloud controller with a DSS that uses limited context modeling (based on available sensor data) and Rule-based Reasoning with fuzzy logic.

- An intermediate configuration that comprises of a physical exercise sensor, a fall sensor, a set of health monitoring sensors measuring pulse and blood pressure, and a set of home sensors that identify the user position and movement. It has a local controller with inbuilt data collection functionality, which forwards the data to the cloud controller. The cloud controller has a DSS with context modeling, fuzzy logic and RBR.

- A complex configuration architecture (Fig.5.3) developed as part of the European Project, CAMI [2]. The CAMI architecture is based on microservices, and has a clean and robust skeleton, onto which several plugin

[2] http://www.aal-europe.eu/projects/cami/

modules can be coupled ensuring modularity and reuse. The major CAMI architecture components are: (i) Sensor unit with multiple sensors, (ii) Data collector unit that collects data from the sensors, (iii) Gateway with intelligent Decision-Support Systems (DSS) for data processing, its local back up, and a DB for storage of data, (iv) Robotic telepresence unit for communication with caregivers, friends, family etc., (v) Mobile phone unit for generation of reminders, and (vi) Cloud Services for redundant data processing (Cloud DSS) and storage (Cloud DB). The CAMI AAL architecture, as a contribution by itself, is presented in Paper B, where we also show its AADL modeling and the initial architecture analysis supported in AADL models developed in OSATE tool, like latency, resource budget and failure.

Modeling the Centralized Architecture in AADL and its Fornal Assurance Framework. For specifying the architecture and its properties, we choose the architecture modeling framework, AADL (Architecture Analysis and Design Language). The generalized AAL architecture along with the DSS is modeled as patterns in AADL (AADL patterns are abstract components). We describe two categories of component patterns in AADL: a) *Atomic Component (AC) Pattern* that do not have hierarchies in terms of sub-components with port interfaces, and b) *Composite Component (CC) Patterns* that has hierarchies with sub-components with port interfaces. We represent the behaviors of the components in the Behavior Annex (BA) of AADL, and the component failures in the Error Annex (EA) of AADL. The EA modeling allows us to efficiently represent the failure and recovery events, which occur via certain probabilistic distributions.

We also show how the AADL patterns can be extended to suit the requirements of the different architecture instantiations mentioned above. For instance, an excerpt of an AADL model of the RBR component (AC) of the DSS (which is a CC), with its interface, BA and EA is shown in Listing 5.1 (The RBR component is common for all the three architecture configurations.) Lines 1-20 define the interface of the RBR component and specify its features, flows, properties and sub-components (in this case, the various data sub-components that do not have port interfaces). The Behaviour Annex (BA) model of the RBR shows a representative scenario of an abnormal high pulse alert generated to the caregiver, and modelled as state transition system in lines 21-28. Lines 29-49 show the error behaviour of the RBR component, modeled as a state transition system in the Error Annex (EA) of AADL. It also shows the failure events causing transient and permanent failures, recovery event (*reset*),

and their distributions.

Listing 5.1: An excerpt from the RBR component in AADL for CAMI

```
1  ——RBR (Component Type +Implementation)———
2  abstract RBR
3  features
4  input: in event data port;
5  output: out event data port;
6  flows
7  Fl : flow path input –> output;
8  properties
9  Dispatch_Protocol => Aperiodic;
10 property_eventgeneration::AperiodicEventGeneration=>1.0;
11 property eventgeneration ::Distribution=> Exponential;
12 property_failure_recovery::FailureRecoveryRate=>1.0;
13 property_failure_recovery::Distribution=> Exponential;
14 Compute_Execution_Time =>1s..1s;
15 end RBR;
16 abstract implementation RBR.impl
17 fuzzy_out_pulse:data  fuzzified_data_pulse;
18 DA: data ADL;
19 u_profile: data user;
20 end RBR.impl
21 —BA—
22 states
23 Waiting: initial complete final state;
24 Operational: state;
25 transitions
26 Waiting –[on dispatch input]–>Operational
27 {if (fuzzyo_pulse=high and DA!= exercising and u_prof =cardiac_patient)
28 {output:= not_caregiver_highpulse}
29 —EA—
30 states
31 Waiting: initial state;
32 Failed_Transient: state;
33 Failed_Permanent:state;
34 LReset: state;
35 Failed_ep:state;
36 events
37 Reset: recover event;
38 TF: error event;
39 PF: error event;
40 Transitions
41 t1: Waiting –[PF]–>Failed_Permanent
42 t2: Waiting –[TF]–>Failed_Transient;
43 t3: Failed_Transient –[Reset]–> {LReset with 0.9,
44 Failed_Permanent with 0.1};
45 t4: LReset–[]–>{Waiting with 0.8, Failed_Permanent with 0.2}
46 properties
47 EMV2::DurationDistribution => [Duration => 1s..2s; applies to Reset;
48 EMV2::OccurrenceDistribution =>[ProbabilityValue => 0.9;
49 Distribution => Fixed;] applies to Reset;
```

To facilitate formal verification, we propose a formal semantics to these AAL specific architecture patterns as a Network of Stochastic Timed Automata (NSTA) that can be model-checked using the UPPAAL model checker or its

statistical extension UPPAAL SMC [6]. An AADL component that we employ in this thesis can be defined as a tuple:

$$AADL_{\text{Comp}} = \langle Comp_{\text{type}}, Comp_{\text{imp}}, EA, BA \rangle, \tag{5.1}$$

where $Comp_{\text{type}}$ represents the component type, $Comp_{\text{imp}}$ represents the component implementation, BA the behavioral annex specification, and EA the error annex. The above definition holds for an AC, but for a CC, we model only the EA showing its composite error behaviour that indicates how the failure of its sub-components affects the CC failure. No separate BA is modeled for CC as its behaviour is already encoded by its AC sub-components. We exemplify the generic definition on the RBR and DSS components in a nutshell. The complete semantics is given in Paper C [6].

$$RBR_{\text{AADL}} = \langle Comp_{\text{type RBR}}, Comp_{\text{imp RBR}}, EA_{\text{RBR}}, BA_{\text{RBR}} \rangle \tag{5.2}$$

On the other hand, the DSS component is a CC defined by its type, implementation and EA, as follows:

$$DSS_{\text{AADL}} = \langle Comp_{\text{type DSS}}, Comp_{\text{imp DSS}}, EA_{\text{DSS}} \rangle \tag{5.3}$$

Formal encoding of AADL components as NSTA. An AADL atomic component (AC), formally defined by the tuple: $AC = \langle Comp_{\text{typeAC}}, Comp_{\text{implAC}}, EA_{\text{AC}}, BA_{\text{AC}} \rangle$ is encoded as an NSTA as follows: $AC \rightsquigarrow NSTA_{AC} = AC_{\text{iSTA}} || AC_{\text{aSTA}}$, where AC_{iSTA} is the so-called "Interface STA" of AC, which corresponds to $Comp_{\text{typeAC}}$ and $Comp_{\text{implAC}}$, whereas AC_{aSTA} is the "Behavioral STA" that encodes the EA and BA of an AC.

Similarly, an AADL Composite Component (CC), formally defined by the tuple: $CC = \langle Comp_{\text{typeCC}}, Comp_{\text{implCC}}, EA_{\text{CC}} \rangle$ is also a network of two synchronized STA, $CC_{\text{NSTA}} = CC_{\text{iSTA}} || CC_{\text{aSTA}}$, where CC_{iSTA} is the "interface" STA of the CC component, and CC_{aSTA} is the "annex" STA that encodes the information from the error annex in AADL. A nutshell of the AADL encoding as NSTA model is presented in Table 5.2, with the detailed description of the encoding rules to be found in Paper C [6].

We take the example of the RBR component and present its semantic encoding as an NSTA model. Let us assume an RBR component defined by Equation 5.2. We define the formal encoding of RBR as the following network of synchronized STA: $RBR_{\text{NSTA}} = RBR_{\text{iSTA}} || RBR_{\text{aSTA}}$, where RBR_{iSTA} is the "interface" STA of the RBR component and RBR_{aSTA} is the "annex" STA that encodes both the behavior and the error annex information.

Table 5.2 Encoding of AADL Component as STA

AADL comp	**STA**
$Comp_{\text{type}}, Comp_{\text{imp}}$	STA
EA, BA	STA
T_{p}	$Invariant + Guard$
T_{e}	$Invariant + Guard$
Ports	$Variables$ + $Synchronization$
$Data$	$Variable$
A sub-component	STA
EA + BA states	*Locations*
EA + BA transitions	$Edges$
Error events	$Variables$
Distribution of error events	γ
Distribution of aperiodic events	μ

Figure 5.4 depicts the NSTA of the RBR based on the encoding rules.

In Paper C [6], we also show the feasibility of using exhaustive model checking with UPPAAL and simulation-based model checking with UPPAAL SMC. For small models, like that of minimal configuration model, exhaustive verification scales. However, for complex models like CAMI, the only feasible option is to use simulation-based model-checking, that is, statistical model checking that returns a probabilistic guarantee of fulfilling a particular requirement.

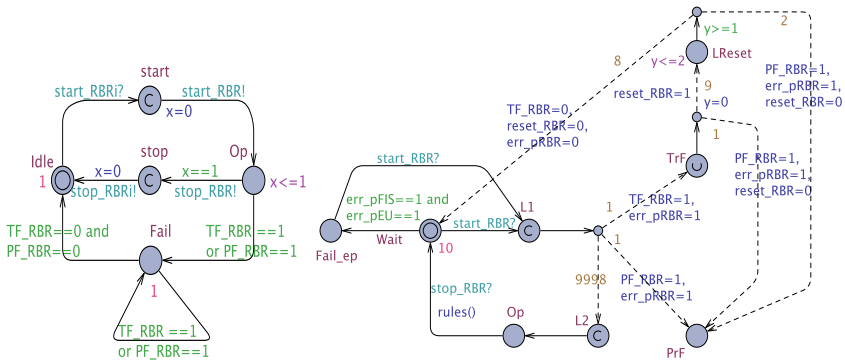
For the CAMI architecture, which is the most complex configuration obtained by instantiating our generic architecture of Fig 5.3, we show the verification of a set of functional and QoS requirements presented.

R1: If the fire sensor detects a fire, then the DSS sends a notification to the firefighters, within 20 s.

R2: If a fall is detected by the wearable or the camera sensor, then the DSS sends a notification to the caregiver, within 20 s.

R3: If there is a pulse data deviation indicating high pulse, the DA is "not exercising", and the user has a disease history of a cardiac patient, then the DSS sends a notification to the caregiver, within 20 s.

R4: If fire and fall are detected simultaneously by the respective sensors, then the DSS should detect the presence of the simultaneous events and send notifications to both the firefighters and the caregiver indicating the presence of both



(a) RBR_{iSTA} (b) RBR_{aSTA}

Figure 5.4: The NSTA for the RBR

events, within 20 s.

R5: The decisions taken by the local DSS are updated in the cloud DSS such that they are eventually synchronized. This requirement relates to the data-consistency requirement of CAMI.

R6: If the local DSS fails, then the cloud DSS eventually becomes active. It corresponds to the fault-tolerance aspect of the CAMI system.

The verification results, by employing UPPAAL SMC, are tabulated in Table 5.3. To check that our CAMI DSS meets its requirements, we employ monitor STA that monitor the sensor values, the respective DSS output, and the corresponding clock. The verification results show that the system satisfies all the functional requirements (R1 to R4) with high probabilities (close to 1) and with high confidence. Requirements R5 and R6 are related the QoS attributes of the CAMI architecture. R5 checks the data consistency of Local DSS and Cloud DSS and requires that the RBR outputs of the local DSS get stored in the case-base of the cloud DSS. This requirement is satisfied with a high probability of [0.99975, 1] and high confidence of 0.998. Query R6 models the fault-tolerance requirement of CAMI. We see from Table 5.3 that the probability of cloud DSS to become activated (R4) is [0.01, 0.04]; this is because it gets activated only when the local DSS has failed and the failure probability of local DSS is between [0.01, 0.04] for a simulation over 1000 time units. However, if the local DSS has failed, we see that the probability of cloud DSS getting activated is very high [0.99975, 1] with a confidence of

0.998, which satisfies our requirement. Most of the requirements are verified with queries that contain terms of the form $A\ imply\ B$, therefore a pre-check of each corresponding “A" being reachable is first carried out.

This contribution is a first attempt to analyze formally an integrated AAL design, including its AI support (context modelling with fuzzy logic and RBR). The described contributions are addressed in Paper B [5] and Paper C [6] and answer Research Questions 2a and 2b.

Table 5.3 SMC analysis results for CAMI Architecture

Req.	Query	Result	Runs
R1	$Pr[<= 1000]([](M\_fire.fire\_alarm == 1)\ imply\ (se\_nw.fire == 1$ and $M\_fire.s1 <= 20)))$	Pr [0.99975,1] confidence 0.998	3868
	$Pr[<= 1000](<>\ (M\_fire.fire\_alarm == 1))$	Pr [0.99975,1] confidence 0.998	4901
R2	$Pr[<= 1000]([](M\_fall.fall\_not == 7)\ imply\ ((se\_w.fall == 1$ or $sd\_nw.data\_val == 1)\ and (M\_fall.s1 <= 20))))$	Pr [0.99975,1] confidence 0.998	3868
	$Pr[<= 1000](<>\ (M\_fall.fall\_not == 7))$	Pr [0.99975,1] confidence 0.998	4901
R3	$Pr[<= 1000]([](M\_pulse.pulse\_not == 3)\ imply\ (110 <= sd\_w.data\_val <= 300\ and\ M\_pulse.FIS\_out == 3\ and\ ADL == 1\ and\ upro.disease\_history == 3$ and $M\_pulse.s1 <= 20))$	Pr [0.99975,1] confidence 0.998	3868
	$Pr[<= 1000](<>\ (M\_pulse.pulse\_not == 3))$	Pr [0.99975,1] confidence 0.998	3868
R4	$Pr[<= 1000]([](\ M\_firefall.fire\_not == 2\ and\ M\_firefall.fall\_not == 2\ imply\ ((se\_w.fall == 1$ or $sd\_nw.data\_val == 1)\ and\ se\_nw.fire == 1$ and $M\_firefall.s1 <= 20))$	Pr [0.99975,1] confidence 0.998	3868
	$Pr[<= 1000](<>\ (Pr[<= 100](<>\ (M\_firefall.fall\_not == 2$ and $M\_firefall.fire\_not == 2))$	Pr [0.99975,1] confidence 0.998	7905
R5	$Pr[<= 1000]([](M\_consistency.stop\ imply\ (RBR\_om == iCBRCC_m)))$	Pr [0.99975,1] confidence 0.998	3868
	$Pr[<= 1000](<>\ (M\_consistency.stop))$	Pr [0.99975,1] confidence 0.998	5777
R6	$Pr[<= 1000]([](INT\_CC.DSSCC\ imply\ PF\_DSS == 1))$	Pr [0.99975,1] confidence 0.998	3868
	$Pr[<= 1000](<>\ (INT\_CC.DSSCC))$	Pr [0.01,0.04] confidence 0.998	2885

5.3 A Multi-agent-based Integrated Architecture for Ambient Assisted Living and its Modeling and Analysis Framework

Driven by the fact that the scalability, adaptability and service accessibility to multiple users are also important concerns that need equal attention as integration and real-timeliness in AAL systems, as the third thesis contribution, we propose an alternative to the previous centralized solution, as a distributed agent-based architecture for AAL systems, which is inspired from the existing architectures in literature. We also propose a modeling and analysis framework for such solutions. The modeling framework uses the core AADL language that we extend with a sub-language called "Agent Annex". The Agent Annex sub-language is formulated by extending the core AADL language class diagrams. The semantics of Agent Annex is defined as Stochastic Transition Systems [14], which can effectively capture the non-deterministic, probabilistic and real-time behaviours of AAL systems. In order to formally verify the system for its requirements, we use the PRISM model checker [25], where the system architecture is modeled as a parallel composition of probabilistic timed automata (PTA) modules. These contributions are be presented in Paper D [12].

The multi-agent AAL architecture that we propose in this thesis (Fig. 5.5) has some important advantages if compared to our centralized AAL solution, in terms of fault-tolerance, scalability, adaptability and accessibility for multiple users. Due to these obvious reasons (although difficult to develop and maintain), distributed architectures are preferred in the AAL community compared to their the centralized counter-parts. However, in our first contribution in which we analyze the existing the agent-based architectures [5], we conclude that the distributed agent-based architectures suffer from an increased overhead for the collective decision making while handling multiple functionalities. Therefore, in our solution, we attempt to reduce such overhead by letting the agents cooperate to ensure intelligent decision making. In our solution, each agent has a particular function, that is, the fire agent deals with detecting fire and raising a notification to the firefighter, the pulse agent detects the pulse deviations and alerts the caregiver, etc. To be able to cooperate efficiently, each agent is equipped with a list of possible dependencies that it can have with other agents. The dependencies can vary dynamically. Each agent in the system is represented with two dependency levels, level 1 and 2, respectively. Dependency Level 1 denotes the list of agents that a particular agent has to cooperate

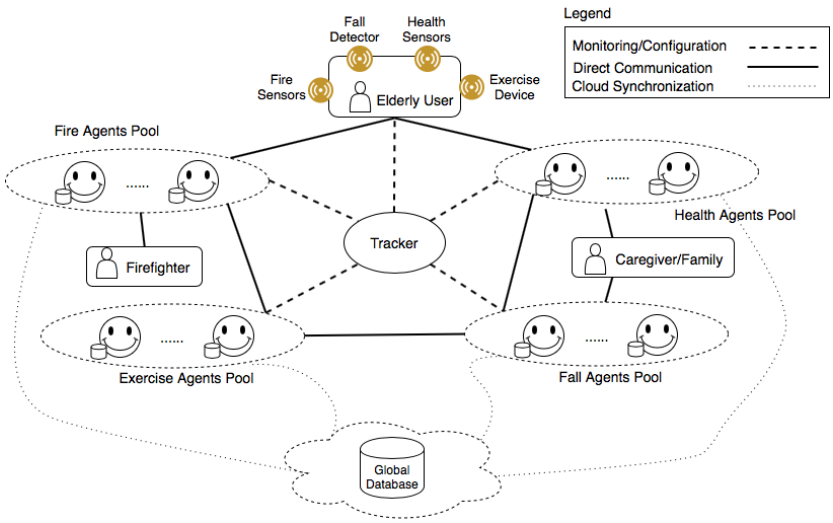


Figure 5.5: A Multi-agent system architecture for AAL systems

with to make an intelligent decision. For instance, in order for the fire agent to notify the user of a fire in the kitchen, it has to cooperate with location agent to identify where the user is. Dependency Level 2 shows the external dependencies that an agent can have in terms of simultaneous occurrence of two events, which does not affect its initial autonomous action directly. For instance, a fire can occur along with a fall. The agents can later synchronize with second-level dependent agents and later compensate for the action taken. In this case, a fire agent compensates its initial action by informing the firefighters that the user has fallen, besides sending the fire notification, such that the firefighters take an immediate action without waiting for a fire event confirmation from the user.

Our multi-agent system (MAS) enables interaction with multiple agents belonging to different categories, ranging from simple reflex agents to complex intelligent agents. In our architecture, we consider only two agent categories: a) *reflex agents* with simple *if-then-else* rules, and b) *intelligent agents* with *reinforcement learning (RL)* [13]. The AAL system described as a MAS contains exercise agents, fall agents, health agents and fire agents (Fig. 5.5). The exercise agent is modelled as an intelligent agent with RL and all other agents are modelled as reflex agents.

Listing 5.2: An excerpt of the fire agent modeling in AADL (interface)

```
1  abstract Fire_Agent1
2  features
3   BA1: requires bus access ACP;
4   BA2: requires bus access SA_comm1;
5  properties
6   Dispatch_Protocol => Periodic;
7   Period => 1 ms;
8   Compute_Execution_time => 2ms..2ms;
9  end agent1;
10 bus ACP ... end ACP;
11 system agent_system ... end agent_system;
12 system implementation agent_system.impl
13 subcomponents
14  A1: abstract Fire_Agent1;
15  Agent_Comm_Protocol: bus ACP;
16 connections
17  BAsys1: bus access Agent_Comm_Protocol <->A1.BA1;
18 end agent_system.impl;
```

We model the proposed architecture in AADL, which we extend with an *Agent Annex* sub-language for specifying agent properties. For simplicity, we show the modeling and analysis on an abstracted version of our MAS architecture consisting of a pulse agent, a fire agent, a fall agent, and an exercise agent each with a redundant copy. These agents cooperate via message-passing. Each agent can accept 2 connections simultaneously from users. We show the case where this system is used by 2 users simultaneously- Jim and Mary.

In the following, we briefly describe our AADL core model and illustrate its Agent Annex. We consider the example of a fire agent as a representative of reflex agents, whereas in Paper D [12], we look at an exercise agent representing the intelligent agents category. The fire agent has the functionality of notifying the firefighter in case of a fire event (with its behavior encoded as if-then-else rules). Listing 5.2 shows an excerpt of the AADL model of a fire agent in our MAS, and a bus component; the Agent Communication Protocol (ACP) models the communication protocol between multiple agents. We assume that the communication protocols defined here work via shared variables. The communication between the agents via tracker is assumed instantaneous, however direct communication between the agents always encounters a delay. The Agent component is modeled as an abstract component in AADL (Lines 1-9), which can be later refined towards a particular hardware or software, based on the application. We also show a system-level representation (Lines 12-18) with its sub-components and their connections defining the communication.

For extending the core AADL with Agent Annex, we first extend the core AADL meta-model [39], specified as UML2 class diagrams with classes for specifying agent behaviours. The meta-model extension is presented in our

Paper D [12]. The *Agent Model Annex* is formulated by extending the AADL abstract classes, *Annex Library* and *Annex Subclause*. The *Annex Library* is used to declare *classifiers* of our Agent Annex in packages. The Annex Library concepts are attached to an AADL model by *using Annex subclause* within a component type or component implementation declaration.

Listing 5.3 shows the syntax of the Annex sub-clause of fire agent attached to its component implementation. It defines a probabilistic transition system with 3 states - $Idle$, $Operational$, and $Fail$, and a clock variable x, and boolean variables $fire$ and the $alarm$. $Idle$ represents the initial state. It also defines a probabilistic transition from state $Idle$. The transition is enabled periodically based on the component's period and it has a probability of 0.999 to reach the state $Operational$, and 0.001 probability to reach the state $Fail$. If the agent reaches the $Operational$ state, it raises a fire alarm in response to the fire event, and takes a transition back to $Idle$, once the component has consumed its execution time (defined by $exec\_time$).

Listing 5.3: An example of Agent Model Annex Subclause attached to Fire Agent

```
1  system implementation fire.agent
2    subcomponents
3    fire_sensor: device f_sensor;
4    annex Agent_Model {**
5      states
6      Idle, Operational, Fail;
7      Idle: initial state;
8      transitions
9      [] state=Idle & x=Period ->0.999:(state '=Operational & x'=0)
10     + 0.001:(state '=Fail);
11     [] state=Idle & x=Period & fire=1 ->0.999:(state '=Operational
12     & x'=0 & fire_alarm '=1)+ 0.001:(state '=Fail &fire_alarm '=0);
13     [] state=Operational & x=exec_time ->state '=Idle & fire_alarm '=0;
14     variables
15     clock x;
16     bool fire;
17     bool fire_alarm;
18   **};
19 end fire.agent;
```

Formal Encoding and Analysis of MAS. An AADL component is defined by the following tuple:

$$AADL_{\text{Comp}} = \langle Comp_{\text{type}}, Comp_{\text{imp}}, AA \rangle, \tag{5.4}$$

where $Comp_{\text{type}}$ is the component type, $Comp_{\text{imp}}$ represents the component implementation, and AA, the agent annex specification [3]. The AA is formally encoded as a tuple:

$$AA = \langle Var, Init, Tt \rangle, \tag{5.5}$$

where: Var represents the set of variables defined in AA, inclusive of state labels, and others representing the sensor data and events, as well as clock variables for specifying the real-time behaviour; $Init$ is the assertion over Var denoting the set of initial states; Tt is the set of state transitions, defined accordingly to follow PRISM's input language syntax defined in Chapter 2.

We also provide a formal encoding of our multi-agent system, including its Agent Annex, in terms of a Stochastic Transition System (STS) [14] as follows:

The AADL component defined by Equation 5.4 is formally encoded as an STS. The MAS architecture is represented as a parallel composition of all the STS modules: $MAS = ||_{i=0}^{n} STS_{modulesi}$, *where* n *is the number of AADL components of the system, excluding data components and bus components, if defined in the system (as variables in the AADL component using them).*

For instance, the fire agent is formally encoded as an STS module, where:

- $V : \{(s1 = 0, fire = 0, fire\_alarm = 0, x = 0), (s1 = 0, fire = 0, fire\_alarm = 0, x = 1), (s1 = 1, fire = 1, fire\_alarm = 1, x = 0), (s1 = 1, fire = 1, fire\_alarm = 1, x = 1), (s1 = 1, fire = 1, fire\_alarm = 1, x = 2), (s1 = 2, fire = 1, fire\_alarm = 0, x = 0)\}$
- $\theta : s \models (s1 = 0 \wedge fire = 0 \wedge fire\_alarm = 0 \wedge x = 0)$
- T is defined by the set of transitions as follows:
  $T : \{\tau 1 : \{(s1 = 0 \wedge fire = 0 \wedge fire\_alarm = 0 \wedge x = period) \longrightarrow (s1' = 0 \wedge fire' = 0 \wedge fire\_alarm' = 0 \wedge x' = 0), P = 1\},$
  $\tau 2 : \{(s1 = 0 \wedge fire = 1 \wedge fire\_alarm = 0 \wedge x = period) \longrightarrow (s1' = 1 \wedge fire' = 1 \wedge fire\_alarm' = 1 \wedge x' = 0), P = 0.999 \cup (s1' = 2 \wedge fire' = 1 \wedge fire\_alarm' = 0 \wedge x' = 0), P = 0.001\},$
  $\tau 3 : \{(s1 = 1 \wedge fire = 1 \wedge fire\_alarm = 1 \wedge x = 2) \longrightarrow (s1' = 0 \wedge fire' = 0 \wedge fire\_alarm' = 0 \wedge x' = 0), P = 1\}\}$

Similarly, all other AADL components are encoded as STS modules, respectively.

[3] Although Agent Annex is specifically tailored to represent agent behaviours, it can also specify the behaviours of other components, like the standard Behaviour Annex.

Listing 5.4: PRISM Model of a Fire Agent

```
1  pta
2  module Fire_agent1
3    s1: [0..2] init 0; //states 0-Idle,1-Operational,2-Fail
4    fire: [0..1] init 0; fire_alert0: [0..1] init 0;
5    x1: clock;
6    invariant
7     (s1=1 => x1<=2) & (s1=0 => x1<=1)
8    endinvariant
9    [1] s1=0 & fire=0 &x1=1-> (s1'=0) &(x1'=0) &(fire_alert0'=0);
10   [2]s1 =0 & fire=1   & x1=1-> 0.999:(s1'=1) & (fire_alert0'=1)
11     & (x1'=0) + 0.001:(s1'=2) &(x1'=0) & (fire_alert0'=0);
12   [3]s1=1  & x1=2  -> (s1'=0) & (x1'=0) &(fire_alert0'=0);
13 endmodule
```

The next step is the formal analysis of the architecture using PRISM model-checker. The STS modules are encoded as PTA modules in PRISM. Listing 5.4 shows the encoding of the fire agent in PRISM. The encoding of STS as PTA modules used by PRISM is straightforward, hence we change only the syntax to match the PRISM input language. The Fire Agent PTA module uses a variable $s1$ to represent the state: $s1 = 0$ (Idle), $s1 = 1$ (Operational), $s1 = 2$ (Fail). There are variables to represent the fire event (*fire: [0..1]*), and raised fall alert (*fire_alert0[0..1]*), where *0* indicates that the event has not occurred, whereas *1* indicates the opposite. Variable $x1$ is a clock variable. The invariants associated with the states (Lines 6-8) depend on the component properties like dispatch protocol and execution time (that are defined at the interface of the AADL component's model). If the component is periodic, the state $Idle$ has the invariant $x1 \leq Period$ associated with it. Similarly, the invariant of state $Operational$ is $x1 \leq Exec\_time$. The transitions (Lines denoted by labels 1-3) follow the transitions definition of the Agent Annex specification of the Fire Agent.

After specifying the architecture as a parallel composition of the PTA modules [12], we verify the the following system requirements with PRISM.

- R1: If a fall occurs due to low pulse, then raise an alert to caregiver indicating *fall due to low pulse* within 20 s.
- R2: If a fire and a fall event occur simultaneously, then raise an alert to both caregiver and firefighter indicating the issue, within 20 s.
- R3: The exercise session is scheduled only if the health agent indicates a normal pulse.

Table 5.4 Verification results

Req.	Query	Result
R1	$filter(forall, fall\_user1 = 1 \& pulse\_user1 \leq 50 \ \& tracker\_fail = 0 \rightarrow P \geq 0.999\ [F((pulse\_alert0\_u1 = 3 \ \| pulse\_alert1\_u1 = 3 \| \ pulse\_alert2\_u1 = 3 \| pulse\_alert3\_u1 = 3)\ \&\ (y \leq 10) \& (fall\_fail = 0)\ \& (pulse\_fail = 0))]$	satisfied
R2	$filter(forall, fall\_user2 = 1 \& fire\_user2 = 1 \& tracker\_fail = 0 \rightarrow P \geq 0.999\ [F((firefall\_alert0\_u2 = 2 \ \| firefall\_alert1\_u2 = 2 \| \ firefall\_alert2\_u2 = 2 \| firefall\_alert3\_u2 = 2) \& (y \leq 10)\ \& (fall\_fail = 0) \ \& (fire\_fail = 0))]$	satisfied
R3	$filter(forall, cal\_notexc\_user1 = 1 \& tracker\_fail = 0 \ \& (pulse\_user1 \geq 60\ \& pulse\_user1 \leq 120) \rightarrow P \geq 0.999\ [F(exc\_sch\_u1 = 1))]$	satisfied
R4	$filter(forall, exc\_sch\_u1 = 1 \& u1\_disease\_history = 1 \ \& u1\_pref = 2 \rightarrow P \geq 0.999\ [F(exc\_u1\_int1 = 2)]$	satisfied
R5	$filter(forall, exc\_sch\_u1 = 1 \& interval = 1\ \& y \leq 5 \ \& pulse\_user1 \geq 200 \rightarrow P \geq 0.999\ [F(exc\_u1\_int2 = 1)]$	satisfied
R6	$filter(forall, fall\_user2 = 1 \& fire\_user2 = 1 \ \& tracker\_fail = 1 \rightarrow P \geq 0.999\ [F((fall\_alert0\_u2 = 2 \ \| fall\_alert1\_u2 = 2 \| \ fall\_alert2\_u2 = 2 \| fall\_alert3\_u2 = 2) \ \& (y \leq 20)\ \&\ (fall\_fail = 0) \& (fire\_fail = 0))]$	satisfied
R7	$filter(forall, fall\_user2 = 1 \& tracker\_fail = 0 \& \ fail1\_fall = 1\ \& fall2\_fall = 0 \rightarrow P \geq 0.999\ [F((fall\_alert2\_u2 = 1 \| fall\_alert3\_u2 = 1)\ \& y \leq 20)]$	satisfied

- R4: The initially suggested exercise is based on user preferences and health condition.
- R5: If any health abnormality is detected in the first sub-session of the exercise, a different set of exercises of lower intensity is prescribed. It should be noted that R1-R5 are safety-critical requirements.

In addition, the system has quality-of-service (QoS) requirements as follows:

- R6: If the tracker fails, the system continues its functionality.
- R7: If one of the agent fails, its function is carried out by the back-up.

The verification results are tabulated in Table 5.4. The requirements are formulated as PCTL queries and the model-checking method is *Digital Clocks*.

Since PRISM, by default, returns the value for the (single) initial state of the model while model checking, we employ *filters* to verify our properties over all states. Requirement $R1$ ensures that if a fall event occurs due to a low pulse for $user1$ (Jim), and the tracker is operational, then the tracker initiates the communication between the respective fall and pulse agents associated with user Jim (the request can be assigned to any of the agent sockets depending on availability), and the probability that one of them sends an alert to caregiver indicating that there is "fall due to low pulse" is greater than 0.999 provided that at least one of the sockets of each agent is functional. Assuming that the communication via tracker takes less time, the requirement is satisfied within 10 time units. Similarly, for $R2$, we verify for $user2$ (Mary) that in case of fire and fall events occurring simultaneously, an alert indicating both events is raised and sent within 10 time units, provided that the tracker has not failed. In case of $R3$, $R4$ and $R5$, we verify the functionality of the exercise agent serving Jim. By $R3$, we establish that the exercise session is scheduled only if the corresponding health agent indicates that the user's pulse level is normal. $R4$ indicates that the initial exercise category is chosen based on user preferences and health condition. By verifying $R5$, we show that if a high pulse deviation occurs during the exercise sub-session, a low intensity exercise is chosen in the next sub-session, irrespective of user preferences. In $R6$, we illustrate a similar function as in $R2$, but assuming that the tracker has failed. In this case, the functionality is met by direct communication between the agents, which takes more time than the communication via tracker (it is shown that this requirement is satisfied within 20 time units). Next, in $R7$, we assume a fall event of $user2$, and one failed fall agent; then, a fall alert is raised and sent to the caregiver by either one of the redundant fall agents. PRISM shows that this requirement is satisfied within 20 time units.

These contributions are contained in Paper D and address Research Questions 2a and 2b.

5.4 Validation with End Users

As a final contribution, we present some initial validation that we have performed on the CAMI architecture (contribution 2), thereby addressing the Research Question 3 formulated in Chapter 4. The research contribution is presented in Paper E. In the analysis presented in the paper, we show the response of 105 senior citizens (55-75 years old) from Romania, Poland and Denmark regarding the CAMI functionalities. With the responses, we give priorities to

health monitoring, fall detection, supervised physical exercises and vocal interaction. We formulate a smaller implemented version of the initial CAMI architecture, presented in Paper B [5]. We also show an initial analysis results of these implemented functionalities by carrying out a set of tests in laboratories by involving different users.

Health-monitoring functionalities: In CAMI, we offer a set of health monitoring functionalities that allow us to monitor blood pressure, heart rate, blood glucose, etc. In addition, CAMI also employs fall detection sensors to identify falls of the elderly and raise timely alerts. Among the respondents, 59% consider the graphic display of various health measurements (e.g. blood pressure, heart rate, oxygen levels) as an interesting feature. The ability to share health measurements with various doctors is considered useful by 60% of the respondents. CAMI's fall detection functionality employing Vibby wearable is tested in laboratory by employing multiple users.

Physical exercise monitoring: CAMI system implements physical exercise monitoring using two avatars: the training avatar and the avatar of the user. The training avatar performs different physical exercises that the user must reproduce. We also test the suitability of exercise avatars in laboratory with multiple users.

Vocal interaction: CAMI's vocal interaction module comprises of modules for automatic speech recognition , natural language understanding, dialog management, natural language generation, and text to speech synthesis. In order to test our vocal interaction functionality, we devise a variety of text inputs for the user to interact with the CAMI system and test the system responsiveness.

A plan for extensive field trails in user homes has been devised and the questionnaires has been set up, and this will be accounted in our future work.

Chapter 6

Related Work

In recent years, there has been a lot of research in the field of AAL, due to to the need for supporting an increased elderly population [4]. In this section, we describe some of the prominent AAL solutions with respect to their software architecture models and and compare the formal approaches where they exist with our solutions.

6.1 Software Architecture Models for AAL

A literature study on existing AAL architectures shows that there are certain architecture types that address the construction of integrative AAL applications (i.e., those that focus on creating a holistic user experience, not just the development of a specific functionality such as health data management or social interaction) [40]. We have classified them into the following architecture types: Multi-Agent Systems (MAS), Cloud-based systems, and Internet-of-Things (IoT) centric.

Agent-based architectures: These are the most commonly used architectures for AAL applications due to their flexibility, autonomy, adaptability, better response and service continuity due to the distributed nature [41, 42]. The agents are autonomous processing entities and can be local and/or cloud based. Some examples of healthcare frameworks that rely on a distributed agent architecture are proposed by Pez et al.[43], Sernani et. al [44], and Tapia et. al[38]. However, the agent based architectures also have some drawbacks, for instance, restricted communication protocols for agent communication and the delay

overhead in taking a collective decision [41].

Cloud-based AAL solutions: There are many AAL solutions that leverage the potential of cloud computing for context modeling [45, 46, 47] intelligent decision making, and use it as a data store [37].

Although cloud-based solutions are scalable, cost-effective, reusable, adaptable, and extendable, the sole processing with cloud cannot guarantee strict hard real-time properties, and the system fails completely in the absence of Internet.

IoT architectures: IoT technology is now getting widely utilized in the field of AAL owing to its technological advancements. The IoT concept of communication (i) between smart objects, (ii) smart objects and people, and (iii) among people themselves, are widely exploited in the field of AAL, thereby providing connectivity, context-awareness and adaptivity [48]. There are also approaches to integrate the autonomous behavior of agent-based systems with IoT technology [49, 50, 51].

Although AAL systems based on IoT offer high flexibility, adaptability, the system depends only on the availability of the Internet for operation, which can lead to a complete failure of such systems in places where Internet connectivity is meager.

In our work, we propose two architecture solutions using the concepts from these existing architecture solutions:

- A centralized solution which uses the cloud for data processing and storage (Paper B [5], Paper C [6]).

  Our architecture is designed with local processing along with cloud processing. In order to overcome the lack of real timeliness property of cloud, we allow hard real-time functionalities to be handled by the local processor. We also overcome the sole dependency of Internet for the operation of our system by a switch which can select using either GSM and Internet communication options.

- A distributed architecture solution with agents and cloud support [Paper D [12]].

  In this architecture, we use cloud as a data store, and for the long-term processing of the stored data. Our architecture eliminates the disadvantages of existing agent systems by providing an efficient way to deal with agent cooperation for intelligent decision making, in real-time.

The backbone of any AAL solution lies in its ambient intelligence utilized for context awareness and intelligent decision making [52]. Many studies have

progressed in this aspect; some of them utilize AI solutions like case-based reasoning [53, 52], fuzzy-logic-based reasoning [54, 55] etc. Based on our preliminary studies [3, 5], we conclude that there is room for improving existing AAL solutions in terms of flexibility and continuity of use, range of provided services, as well as incorporating user preferences into the design, for higher acceptance. In addition, by incorporating multiple AI intelligence algorithms to tackle different scenarios, one can effectively improve the intelligence offered in such solutions. In this thesis, we also propose a DSS architecture combining multiple AI techniques to support enhanced reasoning (Paper C [6]) .

6.1.1 Formal Modeling and Analysis of AAL Systems

Since AAL systems are complex safety-critical systems, which are dynamic and subjected to an unpredictable environment, it is essential that their behavior is analyzed by using formal techniques, to provide assurance that they meet their requirements.

The use of architecture description languages to specify AAL systems has not been exercised previously, yet this is a common approach in automotive or automation systems. In order to specify agent-based systems, there are other approaches for their modeling, especially based on logic-based formalisms, although not so commonly used in the AAL domain. Some of the most popular ones are AUML [56], extended DESCARTES [57], GAIA [58], SLABS [59], CASL[60], DESIRE [61], dMARS [62], agent-based G-net model [63], concurrent METATEM [64] etc. AUML (Agent Unified Modeling Language) [56] is one of the most widely employed modeling framework for agent-based systems in industry. The advantages include provision of simple graphical design tools that enable non-mathematical designers to use it efficiently. However, AUML specifications are often graphical and lack formal semantics. Therefore, one cannot verify AUML designs formally to guarantee certain assurance, which is a very important factor to be considered in the design of safety-critical systems. The specification language called extended DESCARTES [57], provides an executable specification language for BDI (Belief-Desire-Intention) agents, based on Hoare logic. The language is an extension of the DESCARTES specification language [65] developed for specifying real-time systems. However, the language is targeted to specify only closed-loop BDI agents, and lacks expressions for the self-learning of autonomous agents. GAIA [58] is one of the initial approaches for agent-oriented design and analysis. However, the lack of a formal specification language, and the inability

to model dynamic and open systems are its major drawbacks. SLABS (Specification Language for agent Based Systems)[59] is one of the popular agent specification languages that specifies the notion of agents, environment and multi-agent systems, and communication between agents. However, SLABS lacks an executable framework like DESCARTES and fails to express high-level agent properties like self-learning.

CASL (Cognitive Agent Specification Language) [60] specifies agents with mental attributes, knowledge, beliefs, and goals. For the formal specification, it considers the action theory defined by situation calculus. However, limited expressiveness due to the employed modeling notations, and difficulty in specifying complex multi-agent systems are the major drawbacks of CASL. DESIRE [61] is a modeling framework developed to specify multi-agent systems, which allows user to specify various intra-agent and inter-agent functionalities. However, DESIRE lacks a formal language to represent the agents, and does not specify various agent properties, such as beliefs, desires, intentions, commitments etc. dMARS (Distributed Multi-Agent Reasoning System) [62] is based on a Procedural Reasoning System (PRS) [66] for modeling BDI agents. However, the approach restricts only to BDI agents, does not support agent properties such as agent roles, interactions and message passing and it does not provide tool support for executing agent specifications.

Another type of approach relies on using traditional software engineering formalisms. One such approach proposed by Luck and d'Inverno [67] using the Z-specification language. However, the usage of Z makes the specifications of MAS complex. Moreover, certain aspects of MAS, like reactive behaviour are difficult to specify using Z. In addition, these specifications are not executable so simulation and prototyping are not possible [68]. Hilaire et al. [68] have used a multi-formalism-based approach using Object Z and statecharts, which provides expressiveness to specify agent reactiveness and supports prototyping by simulation. However, the specifications are way too complex to be easily comprehensible. Moreover, the Object Z specifications are not executable. Another interesting work by authors in [69], uses Event-B specifications to specify goal-oriented resilient MAS. The approach is not as complex as Z-language, and can specify different abstractions, and in addition, has a tool support Rodin to develop the specifications, hence can be viewed as complementary to our approach due to the deductive approach for verification.

Few works have considered the specification and formal analysis of agent behavior in architecture description languages [70]. However, it uses complex formal semantics that hinder their usability and extension.

In comparison, we propose solutions that are able to specify AAL sys-

tem architectures that possess autonomy, non-determinism, probabilistic and real-time behaviour. In our work, we choose AADL as modeling framework due to the fact that it is a popular architecture modeling framework used with a practical appeal, also providing tool support [19]. In addition to the rich semantics provided by the language to specify real-time embedded systems, and its mechanisms to carry out initial architecture analysis (schedulability, latency, resource utilization, error analysis, etc.), the language is also extensible with user-defined properties and annex sub-languages. In the AADL modeling framework that we propose in Paper D [12] to specify MAS, we present an annex extension to the core AADL language that can specify the non-deterministic, probabilistic, real-time behaviours of agents along with agent learning, reactiveness, and system fault-tolerance. Although we have currently modeled the specifications of a small-scale MAS architecture comprising of simple reflex agents and other self-learning agents utilizing reinforcement learning, due to the extensibility offered by AADL, we can also extend our proposed sub-language to represent any agent types, and their properties. Moreover, there are also many standardized annexes like the Behaviour Annex (BA) and Error Annex (EA) that are integrated to the core AADL, which can also be utilized for specifying the behaviour of systems according to the needs. We show the usage of AADL's EA and BA for specifying the behaviour of our centralized AAL architecture (Paper C [6]).

For the purpose of formal analysis, we employ model-checking. Unlike theorem proving approaches for the structured formal development of systems [71], model-checking is an automated approach, although it has limitations with respect to state space explosion in case of large models and is less expressive compared to theorem-provers.

To enable model checking our architectural models, we transform the AADL model to timed automata constructs. There have also been approaches to formally verify AADL designs in other domains. The transformation approach from AADL to TA or variants has been already addressed by related work [72, 73, 74]. Although these approaches are automated verification techniques, there is a lack of focus on abstract components/patterns with stochastic properties (like our approach in Paper C [6]). In addition, these approaches also suffer from state-space explosion, therefore they might not scale well with complex AAL designs. Nevertheless, there is interesting research that deals with stochastic properties and statistical model checking for the analysis of extended AADL models. One such example is the work of Bruintjes et al. [75], where the authors have used an SMC approach for timed reachability analysis of extended AADL designs. Although our approach in Paper C [6] also focuses

on linear systems, it is different from the mentioned work in the fact that we focus on abstract components, and also introduce BA modeling for capturing the functional behavior of our modules, specifically for modeling the behavior of intelligent DSS. In their work, Bruintjes et al. use the SLIM Language, which is strongly based on AADL and is specific to avionics and automotive industry, including the error behavior and modes. However, we use the AADL core language with its standardized annex sets (EA and BA) for the architecture specification, thereby enabling the representation of the functional and error behavior with the architecture model. The abstract component-based modeling also brings exensiblity and reusability to our approach. Moreover, the authors only consider the event occurrences or delay variations using uniform or exponential distributions, whereas by employing our user-defined properties, we can also specify other distributions. Furthermore, the approach of Bruintjes et al. only deals with evaluation of time-bounded queries, however we also evaluate properties like reliability, data consistency, etc., besides timeliness. Another interesting work [76], possibly carried out in parallel with our work, employs statistical model checking using UPPAAL SMC to evaluate the performance of nonlinear hybrid models with uncertainty modeled in extended AADL. Although the approach is not specific to the AAL domain, it is promising to specify complex CPS systems considering uncertainties from the physical environment. Unlike our model which uses STA, the authors use Priced Timed Automata (PTA) models. In our work in Paper D [12], we also propose the extension of AADL with the Agent Annex specification, whose semantics we define in terms of Stochastic Transition Systems, that can explicitly capture the non-determinism, probabilistic and real-time behavior of AAL systems. We also specify the formal encoding of our AAL specific AADL constructs as PTA and show exhaustive verification in PRISM.

In the AAL domain, we do not have any evidence for any AAL architectures existing on the market being formally assured, although there has been some research in this direction. Parente et al. provide a list of various formal methods that can be used for AAL systems [77]. Rodrigues et al. perform a dependability analysis of AAL architectures using UML and PRISM [11]. Other interesting works use temporal reasoning [10, 78] and Markov Decision Processes to formally verify the reliability of AAL systems [79]. However, the analysis presented in these approaches address only simple scenarios and are not used to analyze complex behaviors of integrated AAL systems and their decision making capabilities during critical scenarios, unlike the work presented in this thesis, in Paper C [6] and Paper D [12].

Chapter 7

Conclusions and Future Work

In this thesis, we have presented the first research steps towards increasing the support offered to the elderly by providing assured intelligent AAL solutions that integrate most of the functionalities that the users would need to be helped in their daily lives.

First of all, we have surveyed the SOA and SOP of existing solutions and identified that most of them are fragmented, less user-friendly, and lack assurance of their functionality and QoS. We have evaluated the performance of AAL systems by combining functionalities of individualized solutions and identified that they are not sufficient to tackle potential critical scenarios involving multiple events, which need combined analysis for enhanced and safe reasoning.

As a second contribution, we have proposed a generic model of an integrated architecture solution for AAL, with a centralized “brain" (intelligent DSS) and its formal assurance framework. This architecture can be chosen as the integration framework for future AAL systems, if major concerns are the ease of development and maintenance. To carry out architecture analysis, we have represented it at pattern-level using AADL. These representations can be easily instantiated to form specific architecture types. In this thesis, we present three different configurations of the generic model - a simple model, an intermediate model and a complex model. To provide formal verification for the AAL systems, we have formally encoded the AADL model into a Network of

Stochastic Timed Automata (NSTA). We also show the formal modeling and analysis of the simple and complex configuration (CAMI architecture). In case of the simple configuration, the formal model represented as NSTA is verified with the model-checking tool UPPAAL, whereas for the complex configuration, exhaustive model-checking does not scale and hence we use the statistical model-checking tool, UPPAAL SMC, to ensure functional behavior with timeliness, consistency and fault-tolerance. The approach presented paves the way for the development of formally assured future intelligent AAL solutions that integrate multiple functionalities and it can be applied at earlier design stages to capture potential errors that can propagate across the development stages, which may result in significant re-engineering costs. Our architecture description framework (AADL) has a commercially available tool support, OSATE [80] for automated modeling, and provides some preliminary architecture level analysis. It also allows us to model the behavior of the architecture components via behavior annex and encode the probabilities of failures of various components, via the error annex. However, AADL also has its limitations of expressing complex behaviors of algorithms such as CBR, which we have omitted in this work. The analysis approach which we use is exhaustive model checking and stochastic model checking, that is automated via commercial tools called UPPAAL [24], and its extension, UPPAAL SMC [9], respectively. The verification results are specific to our architecture instantiations, however one can use the approach to verify any set of requirements for various architecture types defined by the generic architectural model. It is worth mentioning that the results are derived assuming high reliability of individual architecture components and considering specific values for the periods and execution times. However, taking into account the wide variety of available sensors and other components, we can easily adapt the values to account for requirements of any specific architecture.

Our third contribution is another architecture of integrated AAL system architecture, following a distributed approach with multiple intelligent agents and its formal modeling and analysis framework. If fault-tolerance, scalability, adaptability, and simultaneous access to multiple users are the major considerations, then the second solution outweighs the first-one. For representing our agent-based architecture, we use the same AADL modeling framework, as our first solution. Although MAS specifications based on logics and domain specific languages do exist and are popular, they are mostly limited to specification of properties at the agent level and also do not have tool support (see Chapter 6). AADL, on the other hand, allows us to focus on the component level (here *agents*) and also at the system level (*MAS architecture*) and can effectively

model agents' real-time characteristics. However, since the core AADL and its integrated annexes lack expressiveness to specify the behaviour of multi-agent systems that are non-deterministic, probabilistic and real-time, we have proposed a sub-language extension to AADL, named *Agent Annex*. Our modeling framework of MAS is the core AADL language and Agent Annex. The semantics of the modeling framework is encoded as a Stochastic Transition System. We have also proposed a formal analysis framework for a small-scale MAS architecture that possesses four agent categories and their back-ups using the PRISM model checker that supports exhaustive model-checking of probabilistic models. To enable model-checking in PRISM, the AADL model of the system, including its Agent Annex encoded as Stochastic Transition Systems (STS) is represented as Probabilistic Timed Automata (PTA) in PRISM. Also, since our approach is exhaustive, we provide comprehensive guarantees to the functional and QoS attributes of the system.

As a final contribution, we have also shown some initial validation of the architecture of the first category with respect to various functionalities like fall detection, health monitoring, voice interaction and supervised physical exercises.

Future Work. There are several directions for future research endeavors to fill in the voids in the current state of the framework proposed for analyzing our AAL architectures. The most immediate future work is to provide tool support for our model-transformations, that is, to automate the AADL to NSTA and PTA transformations, respectively. We also plan to integrate the Agent Annex sublanguage to the core AADL. In addition, since the PRISM-based formal analysis framework of MAS is hard to scale in case of a larger system with many and different kinds of agents, we intend to show the feasibility of using other model-checkers, like UPPAAL SMC instead. Although this approach can generate only probabilistic guarantees, it is one of the best suited approach to formally analyze complex cyber-physical systems that do not scale with exhaustive verification. Finally, we also envision to perform a comparison of the two architecture solutions that we have proposed in terms of their formal modeling and analysis framework. We also intend to continue with the system validation of CAMI architecture in real-user scenarios.

Bibliography

[1] Department of Economic and Social Affairs Population Division. World Population Ageing 2015. Technical report, United Nations, NewYork, 11 2015.

[2] Parisa Rashidi and Alex Mihailidis. A survey on ambient-assisted living tools for older adults. *IEEE journal of biomedical and health informatics*, 17(3):579–590, 2013.

[3] Ashalatha Kunnappilly, Cristina Seceleanu, and Maria Lindén. Do We Need an Integrated Framework for Ambient Assisted Living? In *Ubiquitous Computing and Ambient Intelligence: 10th International Conference, UCAmI 2016, San Bartolomé de Tirajana, Gran Canaria, Spain, November 29–December 2, 2016, Part II 10*, pages 52–63. Springer, 2016.

[4] Ruijiao Li, Bowen Lu, and Klaus D McDonald-Maier. Cognitive assisted living ambient system: A survey. *Digital Communications and Networks*, 1(4):229–252, 2015.

[5] Ashalatha Kunnappilly, Alexandru Sorici, Imad Alex Awada, Irina Mocanu, Cristina Seceleanu, and Adina Madga Florea. A Novel Integrated Architecture for Ambient Assisted Living Systems. In *Computer Software and Applications Conference (COMPSAC), 2017 IEEE 41st Annual*, volume 1, pages 465–472. IEEE, 2017.

[6] Ashalatha Kunnappilly, Raluca Marinescu, and Cristina Seceleanu. A Model-Checking-Based Framework For Analyzing Ambient Assisted Living Solutions. Mälardalen Real-Time Research Centre, Mälardalen University, March 2019.

[7] Peter Feiler. Open Source AADL tool environment (OSATE). In *AADL Workshop, Paris*, pages 1–40, 2004.

[8] Alexandre David, Dehui Du, Kim G Larsen, Axel Legay, Marius Mikučionis, Danny Bøgsted Poulsen, and Sean Sedwards. Statistical model checking for stochastic hybrid systems. *Electronic Proceedings in Theoretical Computer Science*, pages 122–136, 2012.

[9] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015.

[10] Juan C Augusto and Chris D Nugent. The use of temporal reasoning and management of complex events in smart homes. In *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 778–782. IOS Press, 2004.

[11] Genaína Nunes Rodrigues, Vander Alves, Renato Silveira, and Luiz A Laranjeira. Dependability analysis in the ambient assisted living domain: An exploratory case study. *Journal of Systems and Software*, 85(1):112–131, 2012.

[12] Ashalatha Kunnappilly, Simin Cai, Cristina Seceleanu, and Raluca Marinescu. Architecture modelling and formal analysis of intelligent multi-agent systems. In *14th International Conference on Evaluation of Novel Approaches to Software Engineering*, May 2019.

[13] Richard S Sutton, Andrew G Barto, et al. Introduction to reinforcement learning. 135, 1998.

[14] Luca De Alfaro. Stochastic transition systems. In *International Conference on Concurrency Theory*, pages 423–438. Springer, 1998.

[15] Imad Alex Awada, Oana Cramariuc, Irina Mocanu, Cristina Seceleanu, Ashalatha Kunnappilly, and Adina Magda Florea. An end- user perspective on the cami ambient and assisted living project. In *12th annual International Technology, Education and Development Conference*, March 2018.

[16] Peter H Feiler, Bruce Lewis, Steve Vestal, and Ed Colbert. An overview of the SAE architecture analysis & design language (AADL) standard: a basis for model-based architecture-driven embedded systems engineering. In *Architecture Description Languages*, pages 3–15. Springer, 2005.

[17] RB Frana, J-P Bodeveix, Mamoun Filali, and J-F Rolland. The AADL behaviour annex–experiments and roadmap. In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 377–382. IEEE, 2007.

[18] Julien Delange and Peter Feiler. Architecture fault modeling with the AADL error-model annex. In *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, pages 361–368. IEEE, 2014.

[19] *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language*.

[20] Michael Wooldridge and Nicholas R Jennings. Intelligent agents: Theory and practice. *The knowledge engineering review*, 10(2):115–152, 1995.

[21] Michael Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, 2009.

[22] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

[23] Chih-Han Yu, Justin Werfel, and Radhika Nagpal. Collective decision-making in multi-agent systems by implicit leadership. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 3-Volume 3*, pages 1189–1196. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[24] Kim G Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International journal on software tools for technology transfer*, 1(1-2):134–152, 1997.

[25] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 200–204. Springer, 2002.

[26] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[27] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium*, pages 414–425. IEEE, 1990.

[28] Peter E Bulychev, Alexandre David, Kim G Larsen, Axel Legay, Guangyuan Li, and Danny Bøgsted Poulsen. Rewrite-Based Statistical Model Checking of WMTL. *RV*, 7687:260–275, 2012.

[29] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5):512–535, 1994.

[30] Hilary J Holz, Anne Applin, Bruria Haberman, Donald Joyce, Helen Purchase, and Catherine Reed. Research methods in computing: what are they, and how should we teach them? In *ACM SIGCSE Bulletin*, volume 38, pages 96–114. ACM, 2006.

[31] Jane Webster and Richard T Watson. Analyzing the past to prepare for the future: Writing a literature review. *MIS quarterly*, pages xiii–xxiii, 2002.

[32] Wayne Goddard and Stuart Melville. *Research methodology: An introduction*. Juta and Company Ltd, 2004.

[33] Dawn G Gregg, Uday R Kulkarni, and Ajay S Vinzé. Understanding the philosophical underpinnings of software engineering research in information systems. *Information Systems Frontiers*, 3(2):169–183, 2001.

[34] Mary Shaw. What makes good research in software engineering? *International Journal on Software Tools for Technology Transfer*, 4(1):1–7, 2002.

[35] Visual Paradigm. Visual paradigm for uml. *Visual Paradigm for UML-UML tool for software application development*, page 72, 2013.

[36] Don Batory. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*, pages 7–20. Springer, 2005.

[37] Mobyen Uddin Ahmed, Mats Björkman, and Maria Lindén. A generic system-level framework for self-serve health monitoring system through internet of things (iot). *Studies in health technology and informatics*, 211:305–307, 2015.

[38] Dante I Tapia, Sara Rodrıguez, and Juan M Corchado. A distributed ambient intelligence based multi-agent system for Alzheimer health care. In *Pervasive Computing*, pages 181–199. Springer, 2009.

[39] PA USA Society of Automotive Engineers, Warrendale. AE-AS5506/1, SAE Architecture Analysis and Design Language (AADL) Annex Volume 1, Annex C: AADL Meta-Model and Interchange Formats, 2006.

[40] Martin Becker. Software architecture trends and promising technology for ambient assisted living systems. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2008.

[41] David Isern, David Sánchez, and Antonio Moreno. Agents applied in health care: A review. *International journal of medical informatics*, 79(3):145–166, 2010.

[42] John Nealon and Antonio Moreno. Agent-based applications in health care. *Applications of software agent technology in the health care domain*, pages 3–18, 2003.

[43] Juan De Paz, Sara Rodríguez, Javier Bajo, Juan Corchado, and Emilio Corchado. OVACARE: A multi-agent system for assistance and health care. *Knowledge-Based and Intelligent Information and Engineering Systems*, pages 318–327, 2010.

[44] Paolo Sernani, Andrea Claudi, Luca Palazzo, Gianluca Dolcini, and Aldo Franco Dragoni. Home care expert systems for ambient assisted living: A multi-agent approach. In *Proceedings of the Workshop on The Challenge of Ageing Society: Technological Roles and Opportunities for Artificial Intelligence, Turin, Italy*, volume 6, 2013.

[45] Elarbi Badidi and Larbi Esmahi. A cloud-based approach for context information provisioning. *arXiv preprint arXiv:1105.2213*, 2011.

[46] Alisa Devlic and Klintskog Erik. Context retrieval and distribution in a mobile distributed environment. In *Third Workshop on Context Awareness for Proactive Systems (CAPS 2007)*, 2007.

[47] Abdur Forkan, Ibrahim Khalil, and Zahir Tari. CoCaMAAL: A cloud-oriented context-aware middleware in ambient assisted living. *Future Generation Computer Systems*, 35:114–127, 2014.

[48] Angelika Dohr, Robert Modre-Osprian, Mario Drobics, Dieter Hayn, and Günter Schreier. The Internet of Things for Ambient Assisted Living. *ITNG*, 10:804–809, 2010.

[49] Giancarlo Fortino, Antonio Guerrieri, and Wilma Russo. Agent-oriented smart objects development. In *Computer Supported Cooperative Work in Design (CSCWD), 2012 IEEE 16th International Conference on*, pages 907–912. IEEE, 2012.

[50] Peter Leong and Liming Lu. Multiagent web for the Internet of Things. In *Information Science and Applications (ICISA), 2014 International Conference on*, pages 1–4. IEEE, 2014.

[51] Teemu Leppänen, Jukka Riekki, Meirong Liu, Erkki Harjula, and Timo Ojala. Mobile agents-based smart objects for the internet of things. In *Internet of Things Based on Smart Objects*, pages 29–48. Springer, 2014.

[52] Feng Zhou, Jianxin Roger Jiao, Songlin Chen, and Daqing Zhang. A case-driven ambient intelligence system for elderly in-home assistance applications. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 41(2):179–189, 2011.

[53] Eduardo Lupiani, Jose M Juarez, Jose Palma, Christian Serverin Sauer, and Thomas Roth-Berghofer. Using case-based reasoning to detect risk scenarios of elderly people living alone at home. In *International Conference on Case-Based Reasoning*, pages 274–288. Springer, 2014.

[54] Krasimira Kapitanova, Sang H Son, and Kyoung-Don Kang. Using fuzzy logic for robust event detection in wireless sensor networks. *Ad Hoc Networks*, 10(4):709–722, 2012.

[55] Hamid Medjahed, Dan Istrate, Jérôme Boudy, Jean Louis Baldinger, Lamine Bougueroua, Mohamed Achraf Dhouib, and Bernadette Dorizzi. A fuzzy logic approach for remote healthcare monitoring by learning and recognizing human activities of daily living. In *Fuzzy Logic-Emerging Technologies and Applications*. InTech, 2012.

[56] Bernhard Bauer, Jörg P Müller, and James Odell. Agent UML: A formalism for specifying multiagent software systems. *International journal of software engineering and knowledge engineering*, 11(03):207–230, 2001.

[57] Vinitha Hannah Subburaj and Joseph E Urban. A formal specification language for modeling agent systems. In *Informatics and Applications (ICIA), 2013 Second International Conference on*, pages 300–305. IEEE, 2013.

[58] Michael Wooldridge, Nicholas R Jennings, and David Kinny. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and multi-agent systems*, 3(3):285–312, 2000.

[59] Hong Zhu. SLABS: A formal specification language for agent-based systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(05):529–558, 2001.

[60] Steven Shapiro, Yves Lespérance, and Hector J Levesque. The cognitive agents specification language and verification environment for multiagent systems. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*, pages 19–26. ACM, 2002.

[61] Frances MT Brazier, Barbara M Dunin-Keplicz, Nick R Jennings, and Jan Treur. DESIRE: Modelling multi-agent systems in a compositional formal framework. *International Journal of Cooperative Information Systems*, 6(01):67–94, 1997.

[62] Mark d'Inverno, Michael Luck, Michael Georgeff, David Kinny, and Michael Wooldridge. The dMARS architecture: A specification of the distributed multi-agent reasoning system. *Autonomous Agents and Multi-Agent Systems*, 9(1-2):5–53, 2004.

[63] Haiping Xu and Sol M Shatz. An agent-based Petri net model with application to seller/buyer design in electronic commerce. In *Autonomous Decentralized Systems, 2001. Proceedings. 5th International Symposium on*, pages 11–18. IEEE, 2001.

[64] Marcelo Finger, Michael Fisher, and Richard Owens. Metatem at work: Modelling reactive systems using executable temporal logic. In *Sixth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE-93)*, 1993.

[65] K-Y Sung and Joseph E Urban. Real-time Descartes: A real-time specification language. In *Distributed Computing Systems, 1992., Proceedings of the Third Workshop on Future Trends of*, pages 79–85. IEEE, 1992.

[66] Michael P Georgeff and Amy L Lansky. Reactive reasoning and planning. In *AAAI*, volume 87, pages 677–682, 1987.

[67] Michael Luck, Mark d'Inverno, et al. A formal framework for agency and autonomy. In *ICMAS*, volume 95, pages 254–260, 1995.

[68] Vincent Hilaire, Abder Koukam, Pablo Gruer, and Jean-Pierre Müller. Formal specification and prototyping of multi-agent systems. In *International Workshop on Engineering Societies in the Agents World*, pages 114–127. Springer, 2000.

[69] Linas Laibinis, Inna Pereverzeva, and Elena Troubitsyna. Formal reasoning about resilient goal-oriented multi-agent systems. *Science of Computer Programming*, 148:66–87, 2017.

[70] Flavio Oquendo. π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–14, 2004.

[71] Maksym Bortin, Einar Broch Johnsen, and Christoph Lüth. Structured formal development in Isabelle. *Nordic Journal of Computing*, 13(1/2):2, 2006.

[72] Loïc Besnard, Thierry Gautier, Paul Le Guernic, Clément Guy, Jean-Pierre Talpin, Brian Larson, and Etienne Borde. Formal semantics of behavior specifications in the architecture analysis and design language standard. In *Cyber-Physical System Design from an Architecture Analysis Viewpoint*, pages 53–79. Springer, 2017.

[73] Mohamed Elkamel Hamdane, Allaoui Chaoui, and Martin Strecker. From AADL to timed automaton-A verification approach. *International Journal of Software Engineering and Its Applications*, 7(4), 2013.

[74] Andreas Johnsen, Kristina Lundqvist, Paul Pettersson, and Omar Jaradat. Automated verification of AADL-specifications using UPPAAL. In *High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on*, pages 130–138. IEEE, 2012.

[75] Harold Bruintjes, Joost-Pieter Katoen, and David Lesens. A statistical approach for timed reachability in AADL models. In *Dependable Systems and Networks (DSN), 45th Annual IEEE/IFIP International Conference on*, pages 81–88. IEEE, 2015.

[76] Yongxiang Bao, Mingsong Chen, Qi Zhu, Tongquan Wei, Frederic Mallet, and Tingliang Zhou. Quantitative performance evaluation of uncertainty-aware hybrid AADL designs using statistical model checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(12):1989–2002, 2017.

[77] Guido Parente, Christopher D Nugent, Xin Hong, Mark P Donnelly, Liming Chen, and Enrico Vicario. Formal modeling techniques for ambient assisted living. *Ageing International*, 36(2):192–216, 2011.

[78] using temporal logic and model checking in automated recognition of human activities for ambient-assisted living.

[79] Yan Liu, Lin Gui, and Yang Liu. MDP-based reliability analysis of an ambient assisted living system. In *International Symposium on Formal Methods*, pages 688–702. Springer, 2014.

[80] OSATE-Open Source AADL Test Environment. `http://osate.github.io/`. Accessed: 2018-05-15.

II

Included Papers

Paper A

Chapter 8

Paper A: Do we need an integrated framework for Ambient Assisted Living?

Ashalatha Kunnappilly, Cristina Seceleanu, Maria Lindén.
In Proceedings of the 10th International Conference (UCAmI'16), LNCS, Springer, pages 52–63, Novemeber 2016, Canary Islands, Spain.

Abstract

The significant increase of ageing population calls for solutions that help the elderly to live an independent, healthy and low risk life, but also ensure their social interaction. The improvements in Information and Communication Technologies (ICT) and Ambient Assisted Living (AAL) have resulted in the development of equipment that supports ubiquitous computing, ubiquitous communication and intelligent user interfaces. The smart home technologies, assisted robotics, sensors for health monitoring and e-health solutions are some examples in this category. Despite such growth in these individualized technologies, there are only few solutions that provide integrated AAL frameworks that interconnect all of these technologies. In this paper, we discuss the necessity to opt for an integrated solution in AAL. To support the study we describe real life scenarios that help us justify the need for integrated solutions over individualized ones. Our analysis points to the clear conclusion that an integrated solution for AAL outperforms the individualized ones.

8.1 Introduction

The society is now witnessing a demographic change towards an ageing population. Demographic statistics reports [1] show that the elderly population, that is, people who are older than 60 years of age constitutes about one-fourth of the total population in Europe and is expected to increase in the coming years. A similar ageing trend is also witnessed around the world. The technological boom and the need to support an increasing elderly population have prompted the research community to focus on the field of Ambient Assisted Living (AAL). There are numerous Ambient Intelligent (AI) systems that have been developed to support the elderly in their independent living. In addition, there are significant advances in the field of smart homes, wireless sensor technology, assisted robotics, e-health etc., which have created a breakthrough development in the AAL domain [2] [3]. However, a survey of the existing AAL solutions reveals a potential research gap regarding solutions that integrate all relevant technologies into a common framework.

In practice, almost all of the AAL solutions are found to be fragmented, with limited support of only few integrated functionalities. Nevertheless, it is also possible that one uses various independent systems to build up multiple functionalities. For instance, if an elderly person's home is equipped with an AAL solution that does not have an automatic fall detection system, the user can purchase it separately, as there exist readily available, wearable separate solutions that detect a fall and raise an alarm. This functionality of a fall detection system remains the same whether it is an independent system or part of an integrated framework.

If this is the case, we need to answer an important question - if the individualized solutions can perform their functionality without integration, then do we really need to integrate all of them into a single framework bearing additional cost overheads? This paper focuses on answering such a question. The most obvious reason for a positive answer would be the difficulty encountered in using separate solutions as compared to a single integrated one; however, there might exist a more important reason - the fact that the performance of individualized solutions differ dramatically if they are integrated into a coherent framework, versus the case when they are employed in isolation. We discuss this issue in detail in Section 8.3.

The paper is organized as follows. In Section 8.2, we review some of the prominent AAL solutions with respect to their functionalities. Section 8.3 reasons about the necessity of developing integrated solutions rather than individualized ones. We show this by selecting representative scenarios that we

simulate via sequence diagrams, and check their offline schedules against real-time deadlines. In Section 8.4, we show the functional components of an integrated AAL system by constructing a feature diagram representation. Section 8.5 concludes the paper and gives some directions for future work.

8.2 Literature Survey

In this section, we survey some of the most relevant AAL frameworks developed during the last two decades. The search is restricted to the platforms with multi-functionality support, their availability in the market (or at least at a prototype level) and documented user acceptance. Below, we first list the main functionalities required by AAL systems, and then identify AAL solutions supporting such functionalities. We summarize the results in Table 1, that is, the chosen solutions and their supported functionalities, which justify the rest of the paper.

The functionalities that we have chosen are: health monitoring, fall detection, communication and socialization, support for supervised physical exercises, personalized intelligent and dynamic program management, robotics platform support, intelligent personal assistant that takes orders, gives advice and reminders etc., support for vocal interface, mobility assistance, and home and environment management.

1. *Health monitoring and care:*

   Health monitoring is an important functionality of AAL systems. The parameters to monitor depend on the health status of older adults. The major health monitoring frameworks are inCASA [4], Reaction [5], UniversAAL [6], Diabetic Support Systems, Automated Memory Support for Social Interaction (AMSSI) [2] etc.

2. *Fall detection:*

   The risk of falling is one of the critical hazardous situations that needs to be addressed in an AAL system [7]. Examples of existing solutions include the Tunstall fall alarm, the Bay Alarm sensor etc. The inCASA architecture [4] comprises fall detection for elderly people. The GiraffPlus project [8] uses both the smart phone fall detection application and the Tunstall fall detection sensor.

3. *Communication and social inclusion:*

Communication to health care professionals and social inclusion is another vital functionality of AAL systems. Among the existing frameworks, inCASA [4], Reaction [5], GiraffPlus [8], MobiServ [9] support this functionality.

4. *Supervised physical exercises:*

   Mobility problems are very common for elderly people. Exergames are video games that combine traditional game play with physical activity. The most common sensors available for physical activity detection are Webcamera and Kinect sensors. The Mobiserv [9] framework supports supervised physical exercises.

5. *Personalized, intelligent and dynamic program management:*

   An AAL system should allow the storage of the user's personal data like medication plan, daily, weekly, monthly program planning, exercise planner, record of medical data obtained from sensors, etc. The major frameworks that support this functionality are listed in Table 1.

6. *Robotic platforms support:*

   The service robots like the Pearl, Care-o-Bot, Cero, PR2, Robocare etc. play a significant role in the AAL domain. Another major category is the companion robots, like the robotic baby seal Paro [2]. The major platforms with robotic support include Domeo AAL project [10], GiraffPlus [8], Mobiserv [9] etc.

7. *Intelligent personal assistant:*

   The cognitive abilities of elderly people decrease with age, hence the functionality provided by an intelligent informed friendly collaborator is crucial to AAL systems. Two of the frameworks that support this functionality are inCASA [4], and Reaction [5].

8. *Vocal interface:*

   Most of the elderly regard traditional computer interfaces as overly technical and difficult to use. Hence, support for vocal commands is necessary. There are many existing software systems for speech recognition and synthesis, out of which CMU Sphinx, Julius, Google Speech API are among the most popular. Some of the platforms that support vocal interfaces are listed in Table 8.1.

Table 8.1 Functionalities supported by various AAL frameworks

AAL Platforms	1	2	3	4	5	6	7	8	9	10
inCASA	✓	✓	✓	✗	✗	✗	✓	✗	✗	✓
iCarer	✗	✗	✓	✗	✓	✗	✓	✗	✗	✓
Persona	✗	✗	✓	✗	✓	✗	✓	✓	✗	✓
Reaction	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗
UniversAAL	✓	✓	✓	✗	✓	✗	✓	✗	✗	✗
iDorm	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓
Robocare	✗	✓	✗	✗	✓	✓	✓	✓	✗	✓
Aware Home	✗	✗	✓	✗	✓	✗	✗	✗	✗	✓
Mav Home	✗	✗	✗	✗	✓	✗	✓	✗	✗	✓
CASAS	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓
GiraffPlus	✓	✓	✓	✗	✓	✓	✓	✗	✗	✓
MobiServ	✗	✓	✓	✓	✓	✓	✗	✓	✗	✓

9. *Mobility assistance:*

   Many of the elderly lack the ability to move independently and require mobility support. There are many smart wheel chairs devised for this purpose, like NavChair, Wheelesley, VAHM, PerMMa etc. Apart from wheel chairs, mobility scooters and smart vehicles are prominent in this category [2].

10. *Home and environment management:*

    The most recognized smart home systems that take care of one's home, and perform environment management are iDorm, PERSONA (PERceptive Spaces prOmoting iNdependent Aging)), CASAS Smart Home Project, MavHome, and Aware Home Research Initiative (AHRI) [2].

Table 8.1 gives a synthetic account of various AAL frameworks in terms of supported functionalities. By inspecting the table one can notice that none of the platforms that we have reviewed supports all the functionalities that we have selected. This finding gives rise to a straightforward research question:"Are there any critical performance differences in using an integrated framework versus using individual systems side by side, for achieving the desired functionality?". The answer to this question is elaborated in the next section.

8.3 Analysis of Independent vs. Integrated AAL solutions

In the previous section we have established the fact that none of the reviewed frameworks acts as a fully integrated solution for AAL. In order to answer our original question, we proceed to analyzing a real life scenario involving a fire event and a fall event, possibly occurring simultaneously.

The motivation for choosing this scenario is based on actual data on fall and fire incidents. According to statistics, the fall incidents among elderly people over the age of 75 is at least 30% every year, and 40% of them fall more than once, turning this incident into one of the major risk factors for elderly people, which can sometimes lead to death [11]. Besides falls, the number of fire incidents occurring at home has also been increasing at an alarming rate. According to the report given by Nation Fire Protection, the number of fire incidents reported in 2013 is 369.500, causing 2755 civilian deaths [12]. The fire and fall incidents statistics implies that there is a high likeliness of both events happening at the same time.

We perform our behavioral analysis of solutions by modeling the sequence of message exchanges in the automatic fire and fall detection systems, and by simulating the execution traces of the resulting sequence diagrams in Visual Paradigm [13]. Fire and fall events have hard deadlines associated with their resolution, that is, if a proper timely action is not guaranteed, they will have catastrophic consequences. Therefore, we add response times to individual messages in the sequence diagrams, and thereby calculate the response times of fire detection and fall detection systems, respectively. Next, we analyze the actions schedule of each independent system by using offline scheduling, which is the most suited type of scheduling for hard real time applications [14].

8.3.1 Sequence Diagrams and Schedule Analysis

In this paper, we aim our analysis to understanding the behavior of automatic fire detection and fall detection systems when:

1. Both systems are independent, and:
   (a) Fire and fall events occur at different times.
   (b) Fire and fall events occur simultaneously.
2. Both systems are integrated into a common framework, and:

(a) Fire and fall events occur at different times.

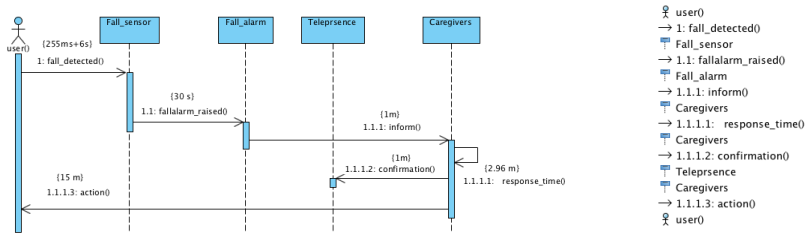
(b) Fire and fall events occur simultaneously.

We also annotate duration constraints to the message interactions in both scenarios, in their respective sequence diagrams. A fall event is usually detected by various feature extraction techniques and fall detection algorithms running in the fall sensor [7]. Most of the fall sensors take approximately 255 ms to get activated on the occurrence of a fall event [15], and take 5 to 6 s to detect a fall [16]. Some of the fall sensors available in the market can raise a fall alarm at 30 s after the detection of a fall [17].

Let us assume that the fall alarm is communicated to a caregiver via an automatic call that takes about 1 min. The average time for the caregiver to respond to the fall alarm call is 2.96 min [18]. The caregiver can validate the fall through a telepresence system (if available), in order to reduce the risk of false alarms. This timing constraint is a variable based on the design aspects of the system. Let us assume that validation takes another 1 min. In case of a real fall, the caregiver should provide assistance immediately. The time taken for providing the required assistance varies, and depends on many factors like the distance from the hospital to the patient's house, type of action taken, etc. For our analysis, we assume a fair time of about 15 min during which proper medical care should reach the person who has fallen.

The fire sensors do not fall under the category "wearable", and hence the system's performance depends on a variety of physical factors like the place of installation, height of installation, type of fire etc. There are now various categories of fire sensors such as ionization sensors, photoelectric sensors, and dual sensors. In this paper, we analyze the performance of a photoelectric sensor installed in a bedroom of a two storey house, which can detect a fire due to flaming, and raise an alarm within 54 s [19]. Let us split this into 2 actions: fire detection within 45 s, and alarm raising within 9 s from detection, in order to carry out a similar analysis like for the fall event. The fire alarm is sent to the firefighter in 30 s [20]. Due to a high number of false alarms, there is a need for confirming the alarm before taking an action. This is usually achieved via telephone call confirmations. All these actions take about 1 min [20]. For detailed analysis, we split this action into 3 subactions: 10 s response time of firefighters, 25 s for call confirmation, and another 25 s for confirmation reply. The volunteer firefighters should arrive at the spot within 9 min from fire alarm confirmation. The fire's put out time is 60 s [20], so the total time for the whole response action is 10 min.

Once we have assigned duration constraints to individual messages in each



(a) Sequence Diagram. (b) Trace

Figure 8.1: Message sequence for fall event.

system, we can analyze both independent and integrated solutions, based on sequence diagram simulations, and then construct their offline schedules.

1. *Behavior of independent, automatic fire detection system, and fall detection system, assuming fire and fall events occur at different times.*

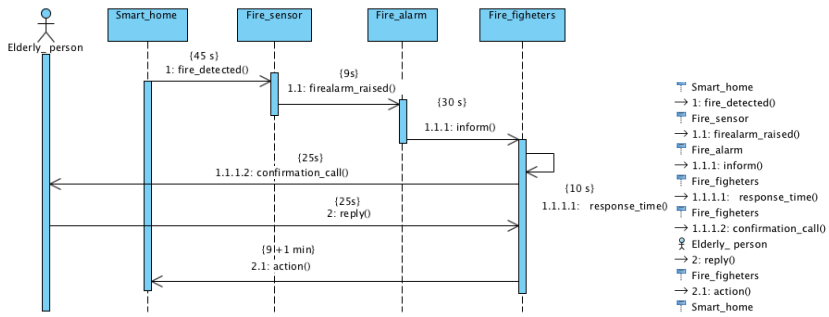
   The sequence diagrams for the individual systems, and their execution traces are described in Figures 8.1 and 8.2, with their respective duration constraints. The duration constraints are assigned as previously discussed in this section. In case of automatic fall detection systems, the total response time is calculated by adding the individual response times of all messages in the sequence diagram of Figure 8.1.

$$R_{fall} = \sum_{i=1}^{6} R_i = 20.56\ min \tag{8.1}$$

   In case of automatic fire detection systems, the response time is calculated in a similar way:

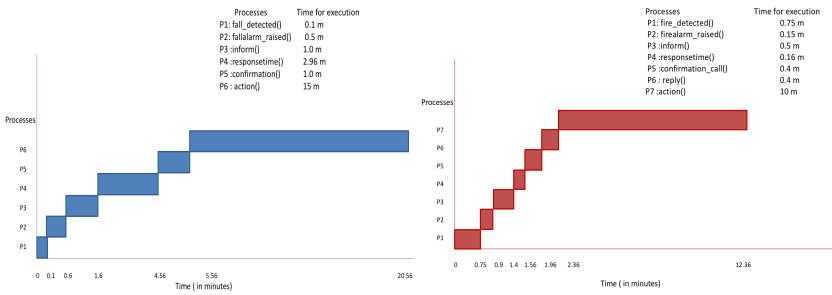
$$R_{fire} = \sum_{i=1}^{7} R_i = 12.36\ min \tag{8.2}$$

   The schedule graphs of the automatic fall detection, and fire detection systems are shown in Figure 8.3. This gives a clear indication of the deadlines associated with each of the events in the context of our analysis: a fall event has to be addressed within 20.56 min to ensure safety, and the fire, once it occurs, has to be extinguished within 12.36 min.



(a) Sequence Diagram. (b) Trace

Figure 8.2: Message sequence for fire event.



(a) Fall event (b) Fire event

Figure 8.3: Offline scheduling of fire and fall event messages.

2. *Behavior of independent, automatic fire detection system, and fall detection system, assuming fire and fall events occur simultaneously.*

   The sequence diagram interactions for independent fire and fall detection systems, with fire and fall events occurring simultaneously is shown in Figure 8.4. In order to better illustrate the simultaneous occurrence of both events, we restrict to a single sequence diagram that shows the interaction of both systems. However, note that these systems are still independent, without any mutual interactions.

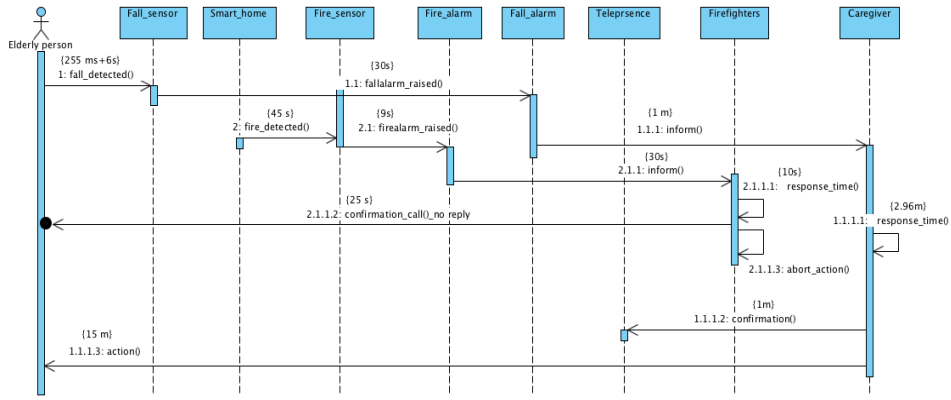


Figure 8.4: Message exchanges of fall event and fire event occurring simultaneously in an independent system.

When the fall and fire events occur simultaneously, the fall event follows the usual sequence of events, but in case of a fire event, the confirmation call of fire event is not answered as the user has fallen (shown as a lost message in the sequence diagram of Figure 8.4. Consequently, it is highly possible that firefighters ignore the fire alarm. Let us imagine the worst case scenario where the fire gets notified to firefighters only when the caregiver arrives to help the fallen individual, that is, the fire event is notified after 20.56 min from start, which is far beyond the deadline of 12.36 min, associated with the fire event. A real catastrophe could occur in this scenario, as the fire is not extinguished in due time.

The deadline miss of the fire event is clearly depicted in the schedule diagram described in Figure 8.6a.

3. *Behavior of an integrated fire detection system and fall detection system, assuming fire and fall events occur at different times.*

   When the fire and fall events occur at disjoint points, the integrated system follows the same sequence of events described by the sequence diagrams in Figures 8.1 and 8.2; the response times for both the events remain the same as in the case of independent systems, and both events

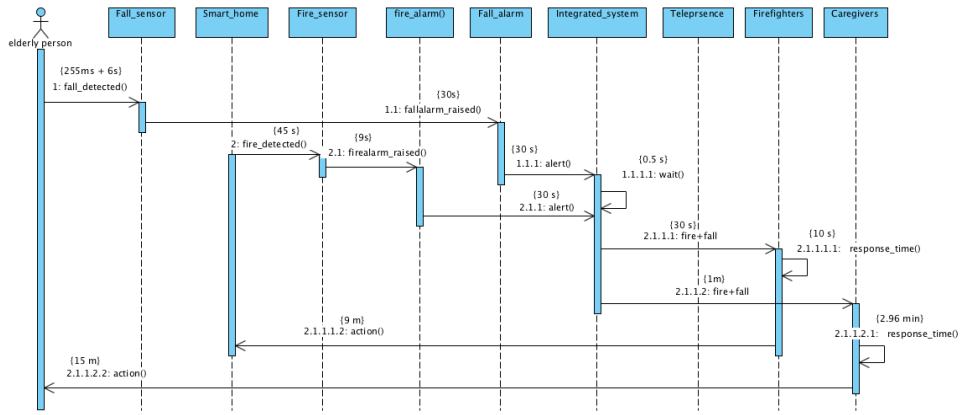


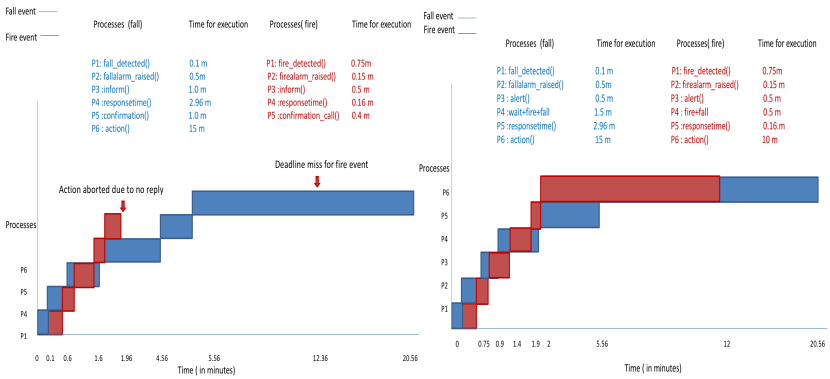
Figure 8.5: Message exchanges of fall event and fire event occurring simultaneously in an integrated system.

are addressed within their deadlines, respectively, as shown in the diagram of Figure 8.3.

4. *Behavior of an integrated fire detection system and fall detection system, assuming fire and fall events occur simultaneously.*

   We have seen earlier that if both fire and fall events occur simultaneously in independent systems, the fire event misses its deadline, which might have catastrophic consequences. Let us now check if the integrated system can address this issue. The sequence diagram in this case is shown in Figure 8.5. As described by the scenario, both individual systems are integrated into a common framework, so both fall and fire events are first communicated to the integrated framework now.

There exists a design constraint associated with the integrated system requiring that the system has to wait for an arbitrary time to check whether some other events are activated at the same time. In our case, when the fall event is sent first to the integrated system, it waits arbitrarily for 0.5 s, such that the fire event that occurred at the same point of time also gets accounted for. We can chose this waiting time, depending on the design constraints, such that the system registers multiple events and does not miss individual deadlines associated with the events. In this case, the integrated framework communicates



(a) Fall and fire events occuring simultaneously in independent systems.

(b) Fall and fire events occuring simultaneously in an integrated system

Figure 8.6: Offline schedules of fire event and fall event messages.

to the firefighters and caregiver that there is fire, and a fall event has occurred also. As such, the firefighters prioritize their rescuing action without requiring a confirmation call, thus completing their action well before the associated deadline. Similarly, the caregiver does not spend time with confirmation, as there are two critical events reported at the same time, so he/she takes his/her action within the associated deadline of 20.53 s. The schedule of this scenario is shown in Fig 8.6b.

The scenario that we have analyzed is one among many scenarios that highlight the necessity of an integrated AAL solution. With the help of this scenario, we could clearly identify that there is a significant performance difference between independent and integrated systems, when multiple critical events occur simultaneously. Due to such evaluations, we can generalize and infer that a potential AAL framework that integrates all the functionalities ranging from health monitoring systems to assisted robotic systems is highly essential to be able to tackle multiple simultaneous events. We need a solution that is capable of making a decision by interconnecting and prioritizing the events in case more than one critical situation occurs. In short, such intelligence can be developed only if one analyzes the scenarios collaboratively, which is only possible if one uses an integrated framework of AAL functions.

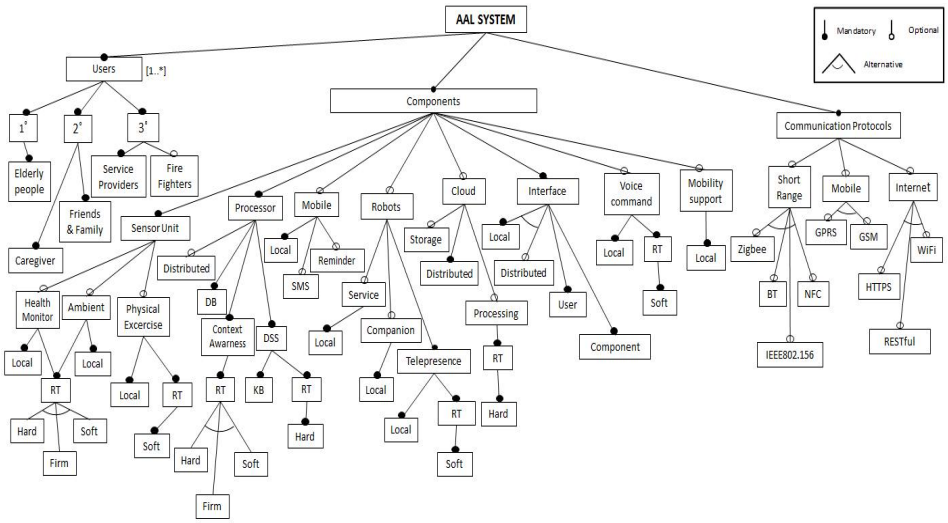


Figure 8.7: Feature Diagram capturing the functions of an integrated AAL system.

8.4 A Feature Diagram of Integrated AAL Functions

Until now we have analyzed whether one should develop or not an integrated architectural solution for AAL. As a first step towards the design of such an integrated architecture for AAL, we capture the functionality features of such a system using a feature diagram representation [21].

The feature diagram depicting the functional components and attributes of an integrated AAL system is shown in Figure 8.7. In a feature diagram, a node with a solid circle represents a mandatory feature of any AAL system. A node with an empty circle represents an optional feature that can be selected by a particular system. Several nodes associated with a spanning curve represent a group of alternative features, from which a feature must be selected for a particular system.

As one can notice, the AAL system is composed of users, components, and communication protocols. The primary users of the system are elderly

adults, but there are also secondary users like the caregivers, family, friends, etc., and also tertiary users like service providers, firefighters, etc. The components include the sensor unit that contains health monitoring sensors (health monitoring), ambient sensors (home monitoring), fall sensors (fall detection), and physical exercise monitoring sensors (supervised physical exercises), mobile phone of the elderly to communicate to external users via SMS, provide reminders, etc., robotic platform support that can also be used as a telepresence system for communication to external users, a private or a public cloud, mobility assistance devices, interfaces, a high end processing unit acting as the core of the AAL system, etc. The processor has the following subcomponents: context awareness module, Decision Support Systems (DSS) with associated Knowledge Bases (KB) and database. The choice of communication protocols is flexible, based on requirements. Each node is also described as local or distributed, with or without Real Time (RT) properties. The diagram in Fig. 7 should help the designer to select the appropriate features of a new AAL system, as well as figure out infeasible combinations of features.

8.5 Conclusions and Future Works

In this paper, we have highlighted the significance of developing an integrated framework for Ambient Assisted Living by analyzing real-life scenarios that justify such a solution. The emergent behavior that arises by integrating the various functionalities like health monitoring, smart homes, physical exercise monitoring systems, robotic platform systems, fall alarms, intelligent friendly collaborator systems, multi modal user interface systems, etc., are essential for ensuring the success of any AAL system. We have also provided a feature diagram representation of the functionalities of an integrated AAL framework, which can serve as design reference of existing or future solutions.

As part of the future work, we plan to propose a fully integrated AAL architecture, which we plan to further model and verify formally against functional and real-time requirements.

Bibliography

[1] Department of Economic and Social Affairs Population Division. World Population Ageing 2015. Technical report, United Nations, NewYork, 11 2015.

[2] Ruijiao Li, Bowen Lu, and Klaus D McDonald-Maier. Cognitive assisted living ambient system: A survey. *Digital Communications and Networks*, 1(4):229–252, 2015.

[3] Parisa Rashidi and Alex Mihailidis. A survey on ambient-assisted living tools for older adults. *IEEE journal of biomedical and health informatics*, 17(3):579–590, 2013.

[4] Andreas P Kapsalis, Georgios Lamprinakos, Konstantinos A Papadopoulos, Dimitra I Kaklamani, and Iakovos S Venieris. The inCASA project: improving the quality of life and social care for the ageing population. *International journal of integrated care*, 12(Suppl1), 2012.

[5] Remote Accessibility to Diabetes Management and Therapy in Operational health care Networks. `http://www.reactionproject.eu`. Accessed: 2016-09-28.

[6] UniversAAL Project. `http://universaal.sintef9013.com/index.php/en/`. Accessed: 2016-09-28.

[7] Yueng Santiago Delahoz and Miguel Angel Labrador. Survey on fall detection and fall prevention using wearable and external sensors. *Sensors*, 14(10):19806–19842, 2014.

[8] Annica Kristoffersson, Silvia Coradeschi, and Amy Loutfi. A review of mobile robotic telepresence. *Advances in Human-Computer Interaction*, 2013:3, 2013.

[9] H Heuvel, C Huijnen, Praminda Caleb-Solly, HH Nap, M Nani, and E Lucet. Mobiserv: A service robot and intelligent home environment for the Provision of health, nutrition and safety services to older adults. *Gerontechnology*, 11(2):373, 2012.

[10] P Rumeau, N Vigouroux, B Boudet, G Lepicard, G Fazekas, F Nourhachemi, and M Savoldelli. Home deployment of a doubt removal telecare service for cognitively impaired elderly people: a field deployment. In *Cognitive Infocommunications (CogInfoCom), 2012 IEEE 3rd International Conference on*, pages 407–412. IEEE, 2012.

[11] Jiangpeng Dai, Xiaole Bai, Zhimin Yang, Zhaohui Shen, and Dong Xuan. Mobile phone-based pervasive fall detection. *Personal and ubiquitous computing*, 14(7):633–643, 2010.

[12] Fire statistics and reports from National Fire Protection. `http://www.nfpa.org/news-and-research/fire-statistics-and-reports/fire-statistics/fires-by-property-type/residential/home-fires`. Accessed: 2016-09-28.

[13] Visual Paradigm for UML. `https://www.visual-paradigm.com`. Accessed: 2016-09-28.

[14] Alan Burns. Scheduling hard real-time systems: a review. *Software Engineering Journal*, 6(3):116–128, 1991.

[15] B Yang, Y Lee, and C Lin. On Developing a Real-Time Fall Detecting and Protecting System using Mobile Device. In *Proceedings of the International Conference on Fall Prevention and Protection, Tokyo, Japan*, pages 151–156, 2013.

[16] Panagiotis Kostopoulos, Tiago Nunes, Kevin Salvi, Michel Deriaz, and Julien Torrent. F2d: A fall detection system tested with real data from daily life of elderly people. In *E-health Networking, Application & Services (HealthCom), 2015 17th International Conference on*, pages 397–403. IEEE, 2015.

[17] Fall Detection Sensors Reviews. `http://medical-alert-systems-review.toptenreviews.com/fall-detection/`. Accessed: 2016-09-28.

[18] Huey-Ming Tzeng and Chang-Yi Yin. Nurses' response time to call lights and fall occurrences. *Medsurg Nursing*, 19(5):266, 2010.

[19] Chris Kasperczyk. Smoke Alarms: Comparing the Differences in Response Times and Nuisance Alarms. Technical report, University of Cincinnati, 2010.

[20] Jennifer D Flynn. NFPA Report: Fire service performance measures, 2009.

[21] Don Batory. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*, pages 7–20. Springer, 2005.

Paper B

Chapter 9

Paper B: A Novel Integrated Architecture for Ambient Assisted Living Systems

Ashalatha Kunnappilly, Alexandru Sorici, Imad Alex Awada, Irina Mocanu, Cristina Seceleanu, Adina Madga Florea
In Proceedings of the IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), pages 465-472, 2017, Turin, Italy, June 2017

Abstract

The increase in life expectancy and the slumping birth rates across the world result in lengthening the average age of the society. Therefore, we are in need of techniques that will assist the elderly in their daily life, while preventing their social isolation. The recent developments in Ambient Intelligence and Information and Communication Technologies have facilitated a technological revolution in the field of Ambient Assisted Living. At present, there are many technologies on the market that support the independent life of older adults, requiring less assistance from family and caregivers, yet most of them offer isolated services, such as health monitoring, reminders etc; moreover none of current solutions incorporates the integration of various functionalities and user preferences or are formally analyzed for their functionality and quality-of-service attributes, a much needed endeavor in order to ensure safe mitigations of potential critical scenarios. In this paper, we propose a novel architectural solution that integrates necessary functions of an AAL system seamlessly, based on user preferences. To enable the first level of the architecture's analysis, we model our system in Architecture Analysis and Design Language, and carry out its simulation for analyzing the end-to-end data-flow latency, resource budgets and system safety.

9.1 Introduction

According to the statistics of the World Population Ageing Report 2015, the world's elderly population is predicted to reach 2.1 billion by 2050, which is more than double of the population of elderly adults in 2015 [1]. The ageing society entails coping with an increased number of diseases, increased healthcare costs, shortage of caregivers [2], etc. Assisted living systems can help in supporting elderly persons in their daily activities and their independent living, with limited risks.

Nowadays, there are numerous Ambient Assisted Living (AAL) solutions available, ranging from a large variety of health monitoring and fall detection sensors, smart homes and assisted robots [3]. However, most current systems are not very effective in critical situations due to not sufficient support of integration of functionalities, difficulty of usage and low acceptance rates [4][5]. One such scenario that supports this claim and that we also analyze in this paper is the occurrence of "fire" and "fall" events simultaneously. When both these events occur together, a safe mitigation of the scenario is achieved only when both these events are communicated to caregivers and firefighters; which is not guaranteed by independent systems working side by side. Assuming that the fire alarm communicated to the firefighters is verified for confirmation by a phone call to the user's home, it follows that the elderly who has fallen that has been communicated to the caregivers only cannot answer in due time, so the fire alarm may be deemed false and discarded, triggering a potential catastrophe. The fact that the existing AAL products do not integrate the modules targeted towards the particular needs of a user, namely health monitoring, home appliances control, report and communication with health professionals, telepresence module etc., offering a solution that could safely resolve potential combined critical situations, also confirmed by model-based behavioral analysis of the system, serves as the motivation for the research presented in this paper. We propose a novel modular architecture for AAL that can be seen as a fully integrated solution, with functionalities selected based on user choices. Our proposed architecture is designed by taking into account the pros and cons of existing prominent AAL architectures in the literature. Since our solution should operate appropriately in critical situations also, it is important that its quality-of-service (QoS) is analyzed. To achieve this, we model the proposed architecture in the Architecture Analysis and Design Language (AADL) [6], and carry out simulations in AADL to estimate the end-to-end flow latency, resource budgets and system safety.

The remainder of the paper is organized as follows. In Section 9.2, we dis-

cuss some of the prominent architectural frameworks developed for AAL and underline their advantages and disadvantages by simulating these architectures in AADL. Section 9.3 describes our integrated architecture, whereas in Section 9.4 we present the run-time architecture model in AADL. The simulation results depicting the end-to-end flow latency, resource budgets and safety analysis in AADL are presented and discussed in Section 9.5. In Section 9.6, we conclude the paper and outline future lines of research.

9.2 Literature Review

Ambient Assisted Living and Ambient Intelligence (AmI) techniques are some of the most researched areas in the past few years due to an increasing amount of elderly population across the world [2]. At present, there are no market solutions that offer a complete integrated solution for AAL. Consequently, in the following, we survey the existing literature on AAL architectural solutions, and identify advantages and shortcomings through AADL simulations in OSATE 2.2.1. AADL has been chosen for architecture analysis due to its architecture-centric and model-based engineering approach, sound specifications and large acceptance in the industry for modeling embedded systems.

This section is organized as follows: in subsection A, we give a brief introduction to AADL and in subsection B, we describe in detail the existing architectures in literature and the results of their AADL simulation.

9.2.1 Architecture Analysis and Design Language

AADL [6] models a system's architecture in terms of hierarchies of components at various levels of abstractions, whose interaction is represented by connections via ports (data, event and event data ports). There are three categories of component abstractions in AADL - *Software, Execution Platform and System*. Application software comprises the process, data, subprogram, thread, and thread group components. The execution platform is made up of computation and communication resources, consisting of processor, memory, bus, and device components. System components are composites that can consist of other systems as well as software or hardware components. The major components are: 1) *Process* - a unit of protected address space, 2) *Data* represents a type, local data subcomponent, or parameter of a subprogram 3) *Thread* - unit of concurrent execution based on various protocols (periodic, aperiodic, sporadic, server and background), 4) *Processor* - a virtual machine that schedules and executes threads, 5) *Memory* - a storage abstraction that can hold data or

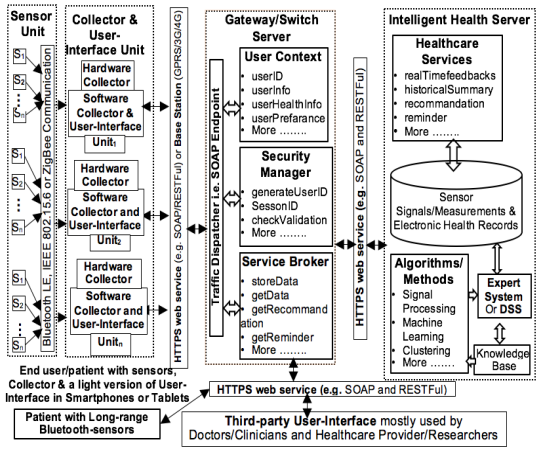


Figure 9.1: ESS-H architecture.

code, 6) *Bus* - a connector abstraction between execution platform components, and 7) *Device* - an abstraction of an active component that an application system can interact with, and a processor executing software that requires access to, via a bus.

9.2.2 Prominent AAL architectures in literature

By examining the AAL literature, we identify some architecture types that address the construction of integrative AAL applications (that is, those that focus on creating a holistic user experience, not just the development of a specific functionality such as health data management or social interaction). In what follows, we investigate two commonly-used architecture types: Cloud based and Multi-Agent System (MAS), showing an example for each.

1. **Cloud-based AAL architectures.**

   In this category, we describe the ESS-H (Embedded Sensor Systems for Health) [7], shown in Fig.9.1. Although the architecture supports multiple functionalities like health monitoring, fall detection, communication to caregiver etc., there is no support for home monitoring (with fire detection systems), robots etc. Hence, inorder to analyze the scenario of simultaneous fire and fall events, the only choice would be to use an independent fire detection system along with the ESS-H. In a related work

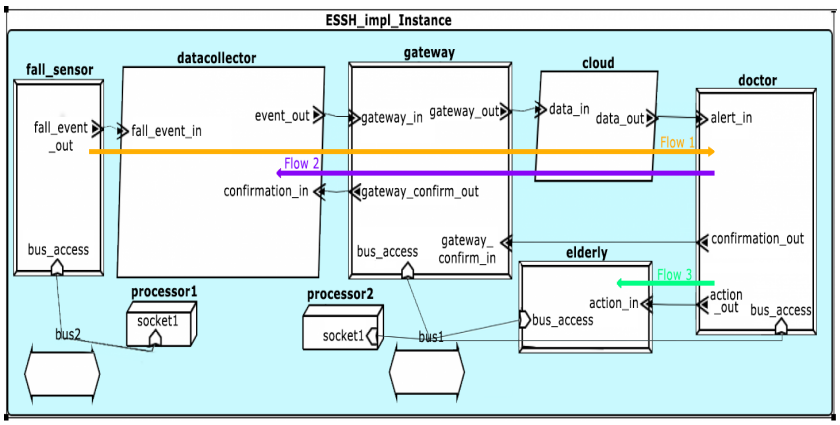


Figure 9.2: ESS-H architecture in AADL describing data flow during fall event.

[5] , the authors argue, based on sequence diagram simulations, that the timing constraint of taking a correct decision could not be met by two independent systems working side by side. However, sequence diagram simulations cannot analyze flows through the components, while considering the sensor sampling times, component execution times and communication bus latencies. Hence, we describe the architecture, systems requirements and flow latency analysis in AADL, for both the ESS-H fall detection system, and the separate fire detection system, respectively.

ESS-H Fall Detection: Architecture Description in AADL. The major components of ESS-H architecture are sensor unit, collector and user-interface unit, the gateway and switch server, and the intelligent health server (IHS), with the servers being cloud based as shown in Fig. 9.1. The data flow during the occurrence of a fall event is described as three end-to-end flows in AADL as shown in Fig. 9.2. Flow 1 describes the data flow from issuing the fall event, until its sending to the caregiver through the data collector (modeled as a process with a thread for analyzing data), gateway (modeled as a device) and cloud (modeled as a process with a thread for communication); Flow 2 is models the caregiver's confirmation of the fall event through the gateway and data collector (which in this case is a tablet); Flow 3 describes the caregiver's action on the elderly in case of a genuine alarm. As we cannot model

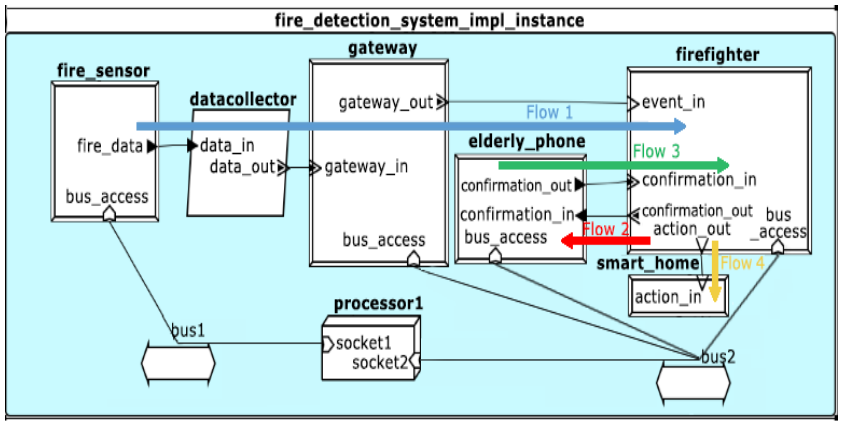


Figure 9.3: Fire detection system architecture in AADL describing data flow during fire event.

human behaviors accurately, we model them as devices with maximum allowed response times.

ESS-H Fall Detection: System Requirements. The system's latency requirements are described in Table 9.1, in accordance with values based on previous work [5], except the split of the human response times into the time needed to register the information and the time needed to take the action. For instance, the maximum allowed response time of the caregiver from the time of being informed of a fall event is 2,96 min [8]; we have divided the response time arbitrarily as 1 min for registering the data (Flow 1) and 1.96 min for the response (Flow 2). The maximum latency values for the end-to-end flows are shown in Table 9.1. For the analysis, we assume periodic sampling of the fall sensor, every 30 sec. All other components are also activated at 30 sec to effectively handle the sensor data. The communication, except that of caregiver's action occurs via two buses - Bluetooth (Latency: 150 to 200 ms) and Internet (Latency: 50 to 100 ms).

ESS-H Fall Detection: Analysis Results The AADL flow latency analysis results are generated based on considering the processing time

Table 9.1 System requirements and flow latency analysis results of ESS-H.

Name	System latency	Max Latency (AADL)
Fall detection system	1233600 ms	928755 ms
Flow1	156255 ms	97755 ms
Flow2	177600 ms	30800 ms
Flow3	900000 ms	800200 ms

Table 9.2 System requirements and Flow latency analysis results of Fire detection system.

Name	System latency	Max Latency (AADL)
Fire detection system	741600 ms	811400 ms
Flow1	91000 ms	69300 ms
Flow2	28000 ms	12000 ms
Flow3	25000 ms	30100 ms
Flow4	600000 ms	700000 ms

of the tasks, processing delay due to queueing, transfer time of information between connections etc. The end-to-end latency analysis results are shown in Table 9.1. All of the three flows meet their maximum end-to-end latencies.

Next, we analyze the independent fire detection system that detects the fire event and communicates it to the firefighters.

Fire Detection: Architecture Description in AADL The fire detection system has a fire detection sensor, a data analyzer module to process the sensor data, a gateway device and a mobile phone to communicate with the firefighters. The fire sensor, gateway, mobile phone and firefighters are modeled as devices, however the data collector is modeled as a process with an associated thread that deals with processing fire sensor data. The AADL model of the fire detection system contains 4 flows: Flow 1 for communicating the fire event to the fire fighter, Flow 2 for confirmation of the fall event, Flow 3 for modeling the response to the confirmation call and Flow 4 for capturing the firefighter's action. The AADL model and associated flows are illustrated in Fig. 9.3.

2. **Distributed agent-based AAL architectures.**

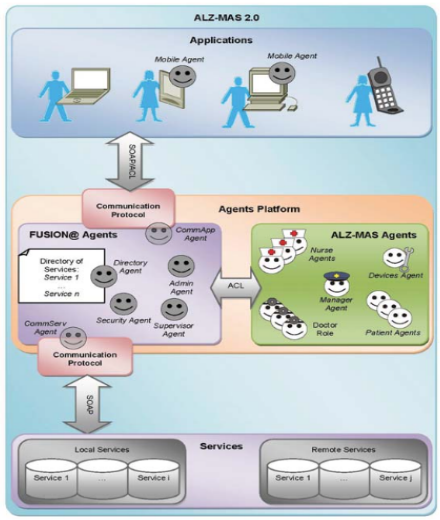


Figure 9.4: A service oriented MAS architecture for Alzheimer health care.

Fire Detection: System Requirements The system's latency requirements are tabulated and described in Table 9.2 [5]. For the flow analysis, we model two different device implementations for mobile phone and firefighter - one showing the normal behavior (in case of Flow 1 and Flow 2) and the other showing the delayed behavior (in case of Flow 3 and Flow 4). Moreover, the Flow 2 is dependent on data from Flow 1, we divide 10 sec response time as 7 sec for registering the fire data and 3 sec for the response. The maximum end-to-end latencies of the flows are thus 1.5, 0.46, 0.4 and 10 min respectively (Table 9.2). We assume that the confirmation call is not answered by the elderly within 5 min. For the analysis, we assume periodic sampling for the fire sensors every 20 sec and all the communication, except firefighter's action occurs via two buses - Bluetooth and Internet with the same prescribed latencies as before.

Fire Detection: Analysis Results The end-to-end flow latency results show that Flow 3 and 4 miss their deadlines. The simulation results are also shown in Table 9.2.

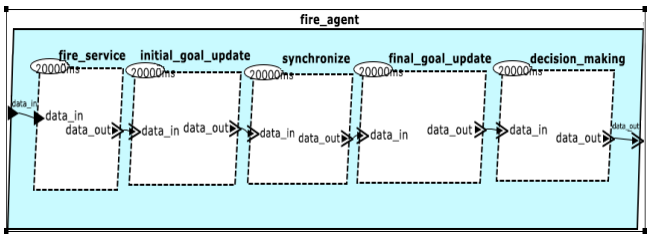


Figure 9.5: Fire agent behavior as AADL threads.

Our analysis shows that the ESS-H solution that consists of fall detection capabilities working side by side with an independent fire detection system might not cover our critical scenario safely, that is, the response actions are not completed by their deadlines. However, later in the paper, we show that if the fire and fall detection capabilities are integrated into the same system, such critical scenarios can be handled in due time. Moreover, the ESS-H solution also runs the risk of single point of failure due to the centralized IHS. The dependency of the architecture on Internet connectivity is high, and there is no local processing of the data, and so the system is exposed to a complete failure in the absence of a working Internet connection, or when the IHS is not able to respond in real-time.

Next, we investigate whether a distributed solution serves the purpose effectively. Thus, we hereby analyze a distributed multi-agent system architecture proposed to support people suffering from the Alzheimer disease [9].

Agent architecture: Architecture Description in AADL The architecture uses a Flexible User and a Service-Oriented multi-ageNt Architecture (FUSION@) [10] and is depicted in Fig. 9.4. Though this architecture does not offer a fully integrated functionality, it can be easily extended to support the intended functionalities owing to its distributed nature. Let us assume that the agent-based architecture contains a fire-agent and a fall-agent to deal with fire and fall events respectively. The actions for a fire agent include detecting the fire event, updating its event list, synchronizing this event with fall agent, updating the event list again and finally taking the decision. We map all these sequences as separate

Table 9.3 Flow latency analysis results of agent based system.

Flows	System Latency Requirements	Max Latency (AADL)
Flow1	91000 ms	170100 ms

threads in AADL as shown in Fig. 9.5. In this system model, we analyze whether fire and fall events are effectively communicated to the firefighter in due time taking into account the agent synchronization and communication delays.

Agent architecture: System Requirements We assume the system's requirements similar to those of the fire detection system described earlier and we assume communication latency of 50 ms to 100 ms for sending synchronization messages.

Agent architecture: Analysis Results The results are tabulated in Table 9.3. As shown, Flow 1 clearly misses its deadline. Hence, even though distributed systems offer higher performance due to resource sharing and has higher reliability and fault tolerance when compared to centralized counterparts, data synchronization becomes a new problem and so does the unpredictable nature of the system (the response times are dependent on the system organization and network load), as shown also by the flow latency analysis. Moreover, if we consider the above system, the solely local deployment of agents makes the system's maintainability difficult. In addition, as the system becomes more distributed, security challenges are higher, and the system's maintainability becomes difficult. As a result of the carried analysis, we opt for a centralized solution, with necessary fault tolerance to deal effectively in real-time, especially in scenarios where multiple events occur together.

Based on the above, we conclude that none of these architectures can be directly used as a framework for building fully integrated AAL systems. Moreover, we have not found any evidence ensuring various QoS attributes. To alleviate such inconveniences, we propose a new architectural solution, named CAMI (Companion with Autonomously Mobile Interface), which is a nominal mix of the studied solutions; described in detail in the following sections.

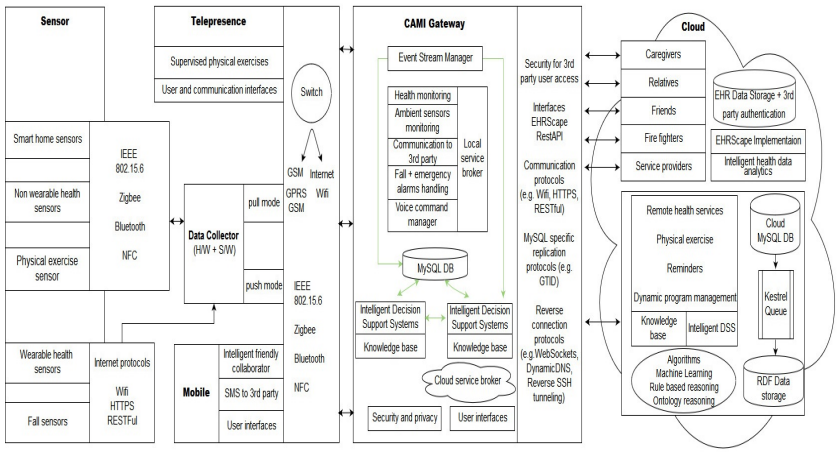


Figure 9.6: CAMI - An integrated architecture for AAL.

9.3 Proposed Architecture

In this section, we describe our novel integrated architecture for AAL, named CAMI[1]. CAMI offers a fully integrated AAL solution by providing services for health monitoring, fall detection, supervised physical exercises, home management and wellbeing as well as telepresence support. CAMI builds an artificially intelligent ecosystem that allows the seamless integration of any number of ambient and wearable sensors, with a mobile robotic platform endowed with multimodal interaction (touch, voice, person detection), including a telepresence robot with manipulation capabilities. As compared to existing solutions, the functionalities that we have chosen to integrate in CAMI are based on user preferences recorded via a multi-national survey with 108 primary and 58 secondary users from Denmark, Romania and Poland.

The architecture is based on the following underlying assumptions: 1) Biometric data must be handled with extreme caution due to privacy laws, 2) The end result of the project has to be feasible from a commercial point of view, 3) Due to the need of a business model, the CAMI system as a whole needs a cloud-based infrastructure.

The CAMI architecture is based on microservices, and has a clean and robust skeleton, onto which several plug-in modules can be coupled ensuring

[1] http://www.aal-europe.eu/projects/cami/

modularity and **reuse**. Two distinct features of CAMI, as compared to other AAL architectures, are: (i) the presence of both **local and cloud-based** processing schemes, and (ii) the **continuity of services** even in the absence of Internet. The CAMI architecture is depicted in Fig. 9.6. The major components of the architecture are: Sensor unit, Data collector unit, CAMI Gateway, Mobile phone unit, Telepresence, and the CAMI Cloud, which are discussed below.

(i) *Sensor unit*: The CAMI system includes various health monitoring sensors, environmental sensors, physical exercise monitoring sensors and fall sensors.

(ii) *Data collector unit*: The interfacing of a wide range of specific sensors/devices with the CAMI ecosystem is achieved by the *Data collector* unit. The unit acts as an intermediate layer aiming at clearly separating the devices from the rest of the CAMI ecosystem, thus increasing its modularity and loose-coupling character.

(iii) *CAMI Gateway*: The CAMI Gateway is a collection of software modules that implement the core infrastructure of the CAMI system. Its purpose is to enable the easy interconnection of the micro services that provide the main functionality of CAMI, like the sensor data collection, intelligent short-term event processing, forwarding of shareable data to CAMI cloud services, etc. At the OS level, there is a switch that can shift the box's operation from Internet to GSM, if needed, in order to ensure that the CAMI system carries out its critical functionalities even in the absence of the Internet connection.

A typical CAMI gateway deployment hosts the following micro services:

1. *Event Stream Manager:* It is a part of the core infrastructure solution enabling message-based interconnection between all the other micro services.

2. *Local Data Storage:* Local data storage offers short-term storage for data collected from sensors, and user information inferred by the decision support algorithms.

3. *Decision Support System (DSS):* The DSS provides a collection of symbolic and data-driven reasoning algorithms that continuously monitor the short-term state of the user (current health status and mood, current and planned daily activities, required reminders, etc).

4. *Voice command manager:* It is offered as a service implementing voice-based interaction with the CAMI system.

5. *Communication with 3rd party health platforms:* This service allows the sharing of selected health measurements and physical exercise sessions with primary and secondary caregivers.

6. *Connection with the CAMI cloud:* Ensures the replication of locally-collected data to the CAMI cloud platform for longer term and higher level processing, as well as the communication with the CAMI cloud to retrieve the results of performed analyses or suggested actions (e.g., context-aware rescheduling of planned activities, and physical exercise recommendations).

(iv) *Robotic telepresence unit*: The CAMI architecture is equipped with an integrated robotic support that is missing from many of the existing AAL frameworks [3]. The robotic teleprsence unit in CAMI can be used for both input and output interactions, and is furthermore capable of actuation.

(v) *Mobile phone unit*: The mobile phone carried by the user acts as an intelligent, friendly collaborator that provides suggestions, advice or reminders. It is also equipped with automatic facilities of sending SMS to third party users like doctors or firefighters in case of emergency situations.

(vi) *Cloud Services*: The CAMI cloud services enable the communication to secondary caregivers (family and friends), healthcare professionals, firefighters, and other CAMI instances. The unit also enables intelligent analysis of user data, collected over a prescribed period of time. Further, it supports the modeling of user data using Semantic Web Technologies (e.g., ontologies for Activities of Daily Living). Finally, cloud services enable the clear/easy administration of each CAMI user account.

9.4 AADL model of CAMI architecture

By their nature, AAL systems are safety-critical real-time systems that need to mitigate possible real-time scenarios of high criticality, which could endanger the life of the elderly. Examples of such situations include scenarios when a person is having a cardiac arrest, or a fire at home etc. Usually, most errors are introduced at design stage of embedded systems, but they are discovered very late, leading to increased rework costs and re-engineering efforts. Therefore, modeling and analyzing AAL architectures at early stages of development can be used to ensure their real-time performance, schedulability, reliability and safety [6]. Consequently, we model the CAMI architecture in the architecture

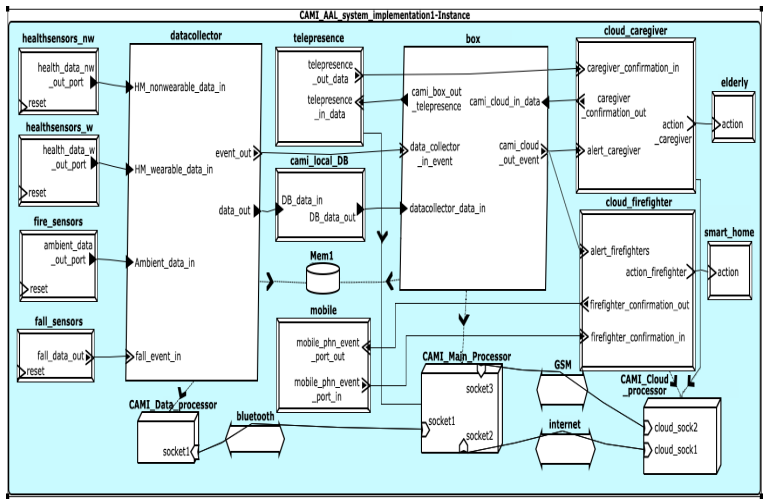


Figure 9.7: CAMI system architecture in AADL showing component interconnections.

description language, AADL [6] using OSATE 2.2.1, and analyze the end-to-end data flow latency of sensor event flows. We also analyze the the system's resource feasibility, its safety and reliability.

The AADL model is shown in Fig. 9.7 and we use this model further to analyze the above attributes. The AADL model of CAMI specifies the communication and data flow between components, through data and event ports. We model all the sensors, the CAMI cloud (caregiver and firefighter), the telepresence, the mobile phone, the smart home and the elderly person as *devices*, and the data collector and the CAMI Gateway as *processes* (Fig. 9.7). The data collector process contains one thread called the "Data_analyzer_module", and the CAMI Gateway contains two threads, "Event_stream" and "DSS". The process components with their threads and port connections for communication are shown in Fig. 9.8 and 9.9. The "Data _analyzer_module" thread in the data collector pre-processes and analyzes the sensor data before it is passed to the CAMI Gateway. All the normal data is passed to the database through the data port of the data collector; however, if any deviations from normal values occur, the data is passed through the output event port, which is then fed to the input event port of the CAMI Gateway. The "Event_ stream" thread in the

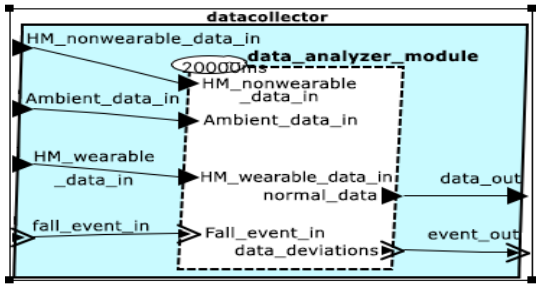


Figure 9.8: Data collector process and its thread.

CAMI Gateway records all the generated events, and passes them to the "DSS" thread for determining the actions in case of events. All the threads except the "Event_stream" are assumed to be periodic, to ensure the continuous monitoring of active events generation, given that the environment is highly dynamic. In comparison, the "Event_stream" is aperiodic and is invoked each time an event occurs. The execution of the modules is controlled by 3 *processors* - "CAMI_Data_processor", "CAMI_Main_processor" and "CAMI_Cloud_processor" with access to various *buses* - "Bluetooth", "Internet" and "GSM".

9.5 CAMI Architecture Analysis in AADL

In the subsections below, we outline the details of various AADL analyses of the CAMI architecture.

9.5.1 Flow latency analysis

In this subsection, we model the end-to-end flows to determine if the CAMI architecture can successfully mitigate critical scenarios involving simultaneous fall and fire events within the respective deadlines. In an integrated system, the occurrences and associated data of concurrent fire and fall events are communicated to both caregivers and firefighters, rendering an immediate rescue action from the firefighters who do not need a phone-based confirmation anymore. The fall event data flow has the same 3 end-to-end flows as for the ESS-H architecture detailed in Section 9.2. Similarly, the fire event data flow is as for ESS-H, adapted to CAMI. The end-to-end latency analysis results are

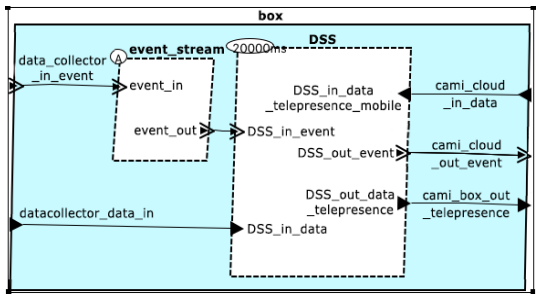


Figure 9.9: CAMI Gateway process and its threads.

Table 9.4 Flow latency analysis results of CAMI system.

Flows	Specified Max Latency	Max Latency (AADL)
Flow1(Fall)	156255 ms	48055.6 ms
Flow2(Fall)	177600 ms	40851.3 ms
Flow3(Fall)	900000 ms	800200 ms
Flow4(Fire)	91000 ms	90400.6 ms
Flow5(Fire)	28000 ms	22501.3 ms
Flow6(Fire)	25000 ms	19001 ms
Flow7(Fire)	600000 ms	402000 ms

summarized in Table 9.4, and the analysis concludes that all the flows meet their respective deadlines.

9.5.2 Resource analysis

AADL has resource analysis plugins also, to analyze resource budgets and allocation during the earlier stages of system development. Resource budgeting can be done for processors, memory, and network bandwidth and can be analyzed to determine whether budgets exceed the allocated sizes (feasibility analysis). We can allocate application components to the execution platform and reconsider the resource budgets in terms of those allocations.

Since our model is designed to illustrate that it could effectively handle the situation with simultaneous occurrence of fire and fall events, only the application components belong to this data flow. Therefore, we associate existing threads to “CAMI_Main_Processor” assigned with a capacity of 200 MIPS

```
                annex EMV2{**
        use types cami::error_library;
                use behavior cami::error_library::simple;

                error propagations
                fall_data_out: out propagation{NoValue,InvalidValue};
                flows
                    ef0: error source fall_data_out{NoValue,InvalidValue};
                    end propagations;
                    component error behavior
                    events
                    Reset: recover event;
                    Fault: error event;
                    transitions
                    t0: Operational-[fall_data_out{NoValue} or fall_data_out{InvalidValue} ]->Failed;
                    t1: Operational-[Fault]->Failed;
                    t2: Failed-[Reset]->Operational;
                    end component;
                    properties
                        emv2::hazards =>
([crossreference =>"N/A";
    failure =>"NoValue";
    phases =>("all");
    description =>"No value from fall detection sensor";
    comment =>"Would impact detecting any fall events";
])
    applies to fall_data_out.novalue;
        emv2::hazards =>
([crossreference =>"N/A";
    failure =>"InvalidValue";
    phases =>("all");
    description =>"Invalid value from fall detection sensor";
    comment =>"Would impact detecting any fall events";
])
    applies to fall_data_out.invalidvalue;
    **};
```

Figure 9.10: Error Annex specification of fall sensor.

Table 9.5 Resource allocation analysis results of CAMI system with single processor.

Components	Resource Capacity	Resource Usage
“CAMI_Main_processor”	200 MIPS	203.5 MIPS
“Mem1”	100 Kbyte	140 Kbyte

and memory capacity of 100 Kbyte, to analyze the respective resource usage. The analysis results shown in Table 9.5 illustrate that the tasks’ resource usage exceeds the processor capacity and memory capacity, hence we add, more processor to our system “CAMI_Data_processor” with capacity 50 MIPS and increase the memory capacity to 150 Kbyte. The process “Data_collector” is associated with “CAMI_Data_Processor”, the process “box” is associated with the “CAMI_Main_Processor” and a memory of 150 Kbyte is associated with the processors. In this case, all the resource budgets are met as shown in Table 9.6.

Table 9.6 Resource allocation analysis results of CAMI system with two processors.

Components	Resource Capacity	Resource Usage
"CAMI_Main_processor"	200 MIPS	200.0 MIPS
"CAMI_Data_processor"	50 MIPS	3.5 MIPS
"Mem1"	150 Kbyte	140 Kbyte

9.5.3 Safety analysis

In the following, we outline the safety analysis of CAMI architecture using the Error Annex (EA) V2 [11]. AADL facilitates different types of safety analysis like the fault hazard analysis (FHA), fault tree analysis (FTA), fault impact analysis etc.

Error Modeling As a first step towards analyzing the safety of CAMI system, we define the error model of the individual components. The CAMI sensor devices are associated with two types of failure: 1) Value Error: When they have no value ("No Value" error) or when they have wrong value ("Invalid Value" error), or 2) Other failure events: E.g., internal failure due to system malfunction ("Fault"). We also define two states of operation of the devices, "Operational" and "Failed". Initially the system is in operational mode, i.e., it performs its required functionality without any errors. If any value error or other fault events occur, the system moves to the failed mode. To return from a failed mode, we define a system self recovery event called "reset". Upon "reset", the system moves back to operational from the failed mode.

Safety Analysis The FHA analysis of the architecture generates an excel report of all potential faults in the system. Fault impact analysis is used show how faults propagate in the system. For this, we assign all the sensor devices as the error flow sources. An example of EA of fall sensor is depicted in Fig. 9.10. Any of the errors in sensors propagate through the data collector and CAMI gateway to the cloud (error sink). FTA also helps us to analyze the failure effects by combining various failure events.

9.6 Conclusions

In this paper, we have proposed an innovative integrated architecture with local and cloud-based processing for AAL systems. In order to validate the performance of our proposed architecture, we have modeled it in AADL, and analyzed the data-flow latency, resource feasibility and system safety. The end-to-end latency analysis has helped in deciding on the combined local and cloud-based centralized architectural solution. The resource analysis in AADL has effectively determined the processor and memory capacities required for executing the application components, facilitating the design decision of resource increase to remove the potential resource usage overflow. Safety analysis in AADL is vital to identify the potential system faults, and analyze their propagation within the system, such that one can recognize what components need back-up and devise error mitigation strategies later.

As future work, we intend to formally verify the CAMI architecture, including the internal behavior of components, especially the DSS behavior under critical scenarios. We envision to produce a full working prototype that will be deployed in the market in the near future.

Acknowledgement

This work has been supported by the joint EU/Vinnova project grant CAMI, AAL-2014-1-087, which is gratefully acknowledged.

Bibliography

[1] Department of Economic and Social Affairs Population Division. World Population Ageing 2015. Technical report, United Nations, NewYork, 11 2015.

[2] Parisa Rashidi and Alex Mihailidis. A survey on ambient-assisted living tools for older adults. *IEEE journal of biomedical and health informatics*, 17(3):579–590, 2013.

[3] Ruijiao Li, Bowen Lu, and Klaus D McDonald-Maier. Cognitive assisted living ambient system: A survey. *Digital Communications and Networks*, 1(4):229–252, 2015.

[4] Hong Sun, Vincenzo De Florio, Ning Gui, and Chris Blondia. The missing ones: Key ingredients towards effective ambient assisted living systems. *Journal of ambient intelligence and smart environments*, 2(2):109–120, 2010.

[5] Ashalatha Kunnappilly, Cristina Seceleanu, and Maria Lindén. Do We Need an Integrated Framework for Ambient Assisted Living? In *Ubiquitous Computing and Ambient Intelligence: 10th International Conference, UCAmI 2016, San Bartolomé de Tirajana, Gran Canaria, Spain, November 29–December 2, 2016, Part II 10*, pages 52–63. Springer, 2016.

[6] Peter H Feiler, Bruce Lewis, Steve Vestal, and Ed Colbert. An overview of the SAE architecture analysis & design language (AADL) standard: a basis for model-based architecture-driven embedded systems engineering. In *Architecture Description Languages*, pages 3–15. Springer, 2005.

[7] Mobyen Uddin Ahmed, Mats Björkman, and Maria Lindén. A generic system-level framework for self-serve health monitoring system through internet of things (iot). *Studies in health technology and informatics*, 211:305–307, 2015.

[8] Huey-Ming Tzeng and Chang-Yi Yin. Nurses' response time to call lights and fall occurrences. *Medsurg Nursing*, 19(5):266, 2010.

[9] Dante I Tapia, Sara Rodrıguez, and Juan M Corchado. A distributed ambient intelligence based multi-agent system for Alzheimer health care. In *Pervasive Computing*, pages 181–199. Springer, 2009.

[10] Dante I Tapia, Sara Rodríguez, Javier Bajo, and Juan M Corchado. FUSION@, a SOA-based multi-agent architecture. In *International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008)*, pages 99–107. Springer, 2009.

[11] Julien Delange and Peter Feiler. Architecture fault modeling with the AADL error-model annex. In *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, pages 361–368. IEEE, 2014.

Paper C

Chapter 10

Paper C: A Model-Checking-Based Framework For Analyzing Ambient Assisted Living Solutions

Ashalatha Kunnappilly, Raluca Marinescu, Cristina Seceleanu. MRTC Report, Mälardalen Real-Time Research Center, March, 2019.
NOTE: This paper is an extended version of the following article: *Assuring Intelligent Ambient Assisted Living Solutions by Statistical Model Checking*. Ashalatha Kunnappilly, Raluca Marinescu, Cristina Seceleanu. In Proceedings of the 8th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), November 2018, Limassol, Cyprus, Springer, pages 457-476.

Abstract

Since modern ambient assisted living solutions integrate a multitude of assisted-living functionalities, some are safety-critical, it is desirable that these systems are analyzed already at their design stage to detect possible errors. To achieve this, one needs suitable architectures that support the seamless design of the integrated assisted-living functions, as well as capabilities for the formal modeling and analysis of the architecture. In this paper, we attempt to address this need, by proposing a generic integrated ambient assisted living system architecture, consisting of sensors, data-collector, local and cloud processing schemes, and an intelligent decision support system, which can be easily extended to suite specific architecture categories. Our solution is customizable, therefore, we show three instantiations of the generic model, as simple, intermediate and complex configuration, respectively, and show how to analyze the first and third categories by model checking. Our approach starts by specifying the architecture, using an architecture description language, in our case, the Architecture Analysis and Design Language that can also account for the probabilistic behavior of such systems. To enable formal analysis, we describe the semantics of the simple and complex categories as stochastic timed automata. The former we model check exhaustively with UPPAAL, whereas for the latter we employ statistical model checking using UPPAAL SMC, the statistical extension of UPPAAL, for scalability reasons. Our work paves the way for the development formally-assured future ambient assisted living solutions.

10.1 Introduction

Elderly people across the world are offered enhanced care via the Ambient Assisted Living (AAL) solutions that support their independent and low-risk living. In order to facilitate the elderly support efficiently and safely, it is often required that these solutions integrate various assisted-living functionalities like health monitoring, home monitoring, fall detection, robotic platform support, communication support, etc. Such integration is extremely beneficial in safety-critical situations, for instance, the case of a fall event occurring due to low pulse, which should trigger sending timely alerts to caregivers, for immediate intervention or else the life of the elderly can be endangered. This requires timely integration of health monitoring (in this case, pulse monitoring) and fall detection functionalities. However, in literature, there are only few architectures, that address the concern of multiple-functionality integration in a timely and robust manner [1, 2]. Due to their critical nature, it is beneficial that such behaviors (especially those emerging due to multiple functionality integration) are analyzed at early stages of development, for instance, at design stage, using formal techniques, to provide some formal guarantees of meeting requirements. There has been some work in this direction, however, the existing frameworks [3, 4] are still in infancy and cannot be used to specify the complete AAL system architecture including its artificial intelligent algorithms, timeliness, reliability, and fault-tolerance attributes.

In this paper, we describe these shortcomings and propose an integrated architecture framework for describing AAL systems and a formal analysis framework that can be employed at the design stages of development. The integrated AAL architecture that we propose supports a range of assisted-living functionalities, like health monitoring, fall detection, reminder services, home monitoring, robotic platform support, etc. and follows the design of common AAL frameworks, with a variety of sensors, data collector unit, user interfaces, intelligent **decision support system (DSS)**, local and cloud processing, etc. Our architecture gives due importance to intelligent decision making by proposing a DSS that employs a mix of artificial intelligent (AI) techniques, like *fuzzy reasoning, rule-based reasoning (RBR) and case-based reasoning (CBR)* for effectively modelling the context space and taking the respective actions based on the current context. The system architecture and its DSS is designed as a generic model that can be customized to fit various categories of architectures, of different complexity. In this work, we show three of such instantiations of our generic model, that is, i) **a minimal configuration** that contains two sensors (pulse and fall), one user interface (a mobile phone), and a cloud controller with a simple DSS system to handle the events from both the sensors, ii) **an intermediate one** with added sensors for blood pressure monitoring, motion detection and exercise monitoring and an enhanced cloud DSS, and iii) **a complex one** comprising wider categories of health monitoring and home monitoring sensors, multiple user interfaces inclusive of robotic telepresence and vocal interactions, and a complex DSS system for handling multiple events simultaneously, and possessing both local and cloud copies for ensuring fault-tolerance via redundancy. The system

architecture, its DSS, and instance models are explained in detail in Section 10.4.

Our contributions also include a modelling and analysis framework proposed for the design-time analysis of complex AAL systems as described earlier. The architecture design relies on the *Architecture Analysis and Design (AADL)* language in which we show the structure and communication between the components of our proposed solution. In AADL, we are able to design the architecture together with the functional and error behavior of the constituting components (Section 10.2.1). Once described, the architecture needs to be analyzed formally for meeting functional and quality-of-service requirements (end-to-end deadlines, fault tolerance, etc.). To enable this, we transform the architecture specifications to a formal model, in our case, the stochastic timed automata (STA) model, that can effectively capture the probabilistic behaviour of AAL components such as random component failures. We demonstrate our formal analysis via two techniques: a) **exhaustive model-checking** using the state of art model checker, UPPAAL, in case of the minimal architecture configuration (for which exhaustive verification scales) and b) **statistical model-checking** with UPPAAL SMC for analyzing the complex model instance [5]. The analysis results are described in (Section 10.7). Although the analysis results are not exact in case of statistical model-checking, these simulation-based methods are sometimes the only choice for reasoning of such complex cyber-physiscal systems (CPS) [6, 7].

10.2 Preliminaries

In this section, we briefly overview AADL, and the other formal notations and tools used for architecture analysis.

10.2.1 The Architecture Analysis and Design Language

AADL [8] is a textual and graphical language in which one can model and analyze a real-time system's hardware and software architecture as hierarchies of components at various levels of abstraction. AADL component categories like ***Application Software*** (*Process, Data, Subprogram, Thread,* and *Thread Group*, etc.), ***Execution Platform***(*Device, Bus, Processor, Memory,* etc.) and ***System*** are used to represent the runtime architecture of the system, however a more generalized representation is possible by specifying a component type as ***abstract.***

AADL allows possible component interactions via *ports/features*, *shared data*, *subprograms*, and *parameter connections*. In AADL, the input/output ports can be defined as: *event ports*, *data ports*, and *event-data ports*. Based on the component interactions, explicit *control flows* and *data flows* can be defined across the interfaces of AADL components by specifying the components as *flow source*, *flow path* or *flow sink*. The components can also be associated with various *properties*, like the *period* and *execution time* and the *dispatch protocol*. The *dispatch protocol* specifies if the component trigger is *periodic* or *aperiodic*.

A component in AADL can be defined by its *type* and *implementation*. The *component type* declaration defines the interface of the component (defining the component category and its interaction points with other components) and its externally observable attributes, whereas the *component implementation* defines its internal structure in terms of its subcomponents and connections between them. In this paper, we distinguish the subcomponents that are composed within a component in *port* interfaces in terms of their port interfaces. For instance, a *data component*, has no interfaces defined in terms of input-output ports, however it can be defined as a subcomponent of another component. We refer such components as *Atomic Components.* However, if a component is composed of another component with port interfaces (like device, thread, abstract, etc.), then a well-defined component hierarchy is identified and we call such components as *Composite Components.*

The AADL core language can be extended via annex sublanguages and user-defined properties. In this work, we employ the standardized annexes of AADL for describing the functional and error behavior of a component, namely the Behavior Annex (BA) [9] and the Error Annex (EA) [10] respectively, which model behaviors as transition systems. The BA state machine interacts with the component interface and represents the system behavior. Given finite sets of states and state variables, the behavior of a component is defined by a set of state transitions of the form $s \xrightarrow{guard,\ actions} s'$, where s, s' are *states*, *guard* is a boolean condition on the values of state variables or presence of events/data in the component's input ports, and *actions* are performed over the transition and may update state variables, or generate new outputs. Similarly, the EA models the error behavior of a component as transitions between states triggered by error events. It is also possible to represent the different types of errors, recovery paradigms, probability distribution associated with the error states and events, and also specify error flows and propagations within the component, and between various components.

In this paper, we focus on *abstract* components that allow us to defer from the run-time architecture of the system. The need for this generic model stems from the fact that in real-world applications like AAL, it is difficult to assign run-time semantics to components before the design matures. These generic component categories can be parametrized, and can be refined later in the design process through the “extends" capability of AADL. AADL allows us to archive these components and reuse them. For this, we partition them into two public packages in AADL, namely *component library* and *reference architecture* [11]. A *component library* creates a repository of component types and implementations with simple hierarchy. It can be established via two packages: (i) *Interfaces Library* comprising generic components like sensors, actuators and user-interfaces (UI), and (ii) *Controller Library* that includes the control logic. The *Reference architecture* creates a repository of components of complex hierarchy, e.g. the top-level system architecture.

10.2.2 Formal Notations and Tools

The formal analysis technique employed in this paper is model checking. We employ two different types of model checking in this paper- 1) exhaustive-model checking using the state-of-the-art model checker UPPAAL, and 2) statistical model-checking, using the statistical extension of UPPAAL model checker, UPPAAL SMC. In the following, we overview the semantics of the input models and the mentioned tools.

10.2.3 Timed Automata and Stochastic Timed Automata

A timed automaton (TA) as used in the model checker UPPAAL is a formal notation for describing real-time systems [12], and is defined by the following tuple:

$$TA = \langle L, l_0, A, V, C, E, I \rangle \tag{10.1}$$

where: L is a finite set of *locations*, $l_0 \in L$ is the *initial location*, $A = \Sigma \cup \tau$ is a set of *actions*, where Σ is a finite set of *synchronizing actions*($c!$ denotes the send action, and $c?$ the receiving action) partitioned into inputs and outputs, $\Sigma = \Sigma_{\mathrm{i}} \cup \Sigma_{\mathrm{o}}$, and $\tau \notin \Sigma$ denotes internal or empty actions without synchronization, V is a set of *data variables*, C is a set of *clocks*, $E \subseteq L \times B(C,V) \times A \times 2^C \times L$ is the set of *edges*, where $B(C,V)$ is the set of *guards* over C and V, that is, conjunctive formulas of clock constraints ($B(C)$), of the form $x \bowtie n$ or $x - y \bowtie n$, where $x, y \in C$, $n \in \mathbb{N}$, $\bowtie \in \{<, \leq, =, \geq, >\}$, and non-clock constraints over V ($B(V)$), and $I : L \longrightarrow B_{dc}(C)$ is a function that assigns *invariants* to locations, where $B_{dc}(C) \subseteq B(C)$ is the set of downward-closed clock constraints with $\bowtie \in \{<, \leq, =\}$. The invariants bound the time that can be spent in locations, hence ensuring progress of TA's execution. An edge from location l to location l' is denoted by $l \xrightarrow{g,a,r} l$, where g is the guard of the edge, a is an update action, and r is the clock reset set, that is, the clocks that are set to 0 over the edge. A location can be marked as *urgent* (marked with an U) or *committed* (marked with a C) indicating that the time cannot progress in such locations. The latter is a more restrictive, indicating that the next edge to be transversed needs to start from a *committed* location.

The semantics of TA is a *labeled transition system*. The states of the labeled transition system are pairs (l, u), where $l \in L$ is the current location, and $u \in R^C_{\geq 0}$ is the clock valuation in location l. The initial state is denoted by (l_0, u_0), where $\forall x \in C,\ u_0(x) = 0$. Let $u \vDash g$ denote the clock value u that satisfies guard g. We use $u + d$ to denote the time elapse where all the clock values have increased by d, for $d \in \mathbb{R}_{\geq 0}$. There are two kinds of transitions:

(i) *Delay transitions*: $< l, u > \xrightarrow{d} < l, u + d >$ if $u \vDash I(l)$ and $(u + d') \vDash I(l)$, for $0 \leq d' \leq d$, and

(ii) *Action transitions:* $< l, u > \xrightarrow{a} < l', u' >$ if $l \xrightarrow{g,a,r} l', a \in \Sigma, u \vDash g$, clock valuation u' in the target state (l', u') is derived from u by resetting all clocks in the reset set r of the edge, such that $u' \vDash I(l')$.

A stochastic timed automaton (STA) refines TA as follows: (i) probabilistic choices between multiple enabled transitions, where the output *probability* function γ may be defined by the user, and (ii) probability distributions for non-deterministic time delays, where the *delay density function* μ is a uniform distribution for time-bounded delays or an exponential distribution with user-defined rates for cases of unbounded delays. Formally, an STA is defined by the tuple:

$$STA = \langle TA, \mu, \gamma \rangle \tag{10.2}$$

The delay density function (μ) over delays in $\mathbb{R}_{\geq 0}$ is either a uniform or an exponential distribution depending on whether the time in location l is bounded by an invarinat, or is unbounded, respectively. With E_l we denote the disjunction of guards g such that $l \xrightarrow{g,o,\text{-}} \text{-} \in E$ for some output o. Then $d(l,v)$ denotes the infimum delay before the output is enabled, $d(l,v) = \text{inf}\,\{d \in \mathbb{R}_{\geq 0} : v + d \vDash E(l)\}$, whereas $D(l,v) = \text{sup}\,\{d \in \mathbb{R}_{\geq 0} : v + d \vDash I(l)\}$ is the supremum delay. If the supremum delay $D(l,v) < \infty$, then the delay density function μ in a given state s is the same is a uniform distribution over the interval $[d(l,v); D(l,v)]$. Otherwise, when the upper bound on the delays out of s does not exist, μ_s is an exponential distribution with a rate $P(l)$, where $P : L \to \mathbb{R}_{\geq 0}$ is an additional distribution rate specified for the automaton. The output probability function γ_s for every state $s = (l,v) \in S$ is the uniform distribution over the set $\{o : (l,g,o,\text{-},\text{-}) \in E \wedge v \vDash g\}$.

In this paper, we use STA to model our AAL system architecture.

10.2.4 UPPAAL and UPPAAL SMC

UPPAAL model checker provides exhaustive model-checking of timed-automata models like the ones overviewed in Section 10.2.2. A real-time system can be modeled as a network of TA (NTA) composed via the parallel composition operator ("||"), which allows an individual automaton to carry out internal actions, while pairs of automata can perform handshake synchronization. The locations of all automata, together with the clock valuations, define the state of an NTA. The properties to be verified by model checking on the resulting NTA are specified in a decidable subset of (Timed) Computation Tree Logic ((T)CTL) [13], and checked by the UPPAAL model checker. UPPAAL supports verification of liveness and safety properties [14]. The queries that we verify in this paper are of the form: i) **Reachability**: $E\Diamond p$ means that there exists a path where p is satisfied by at least one state of the path, and (ii) **Time bounded Leads to**: $p \rightsquigarrow_{\leq t} q$, which means that whenever p holds, q must hold within at most t time units thereafter.

UPPAAL SMC [5], the extension of UPPAAL for statistical model checking, provides means to formally analyze stochastic models. A model in UPPAAL SMC consists of a network of interacting STA (NSTA) that communicate via broadcast channels and shared variables. In a broadcast synchronization one sender $c!$ can synchronize with an arbitrary number of receivers $c?$. In the network, the automata repeatedly race against

each other, that is, they independently and stochastically decide how much to delay before delivering the output, and what output to broadcast at that moment, with the "winner" being the component that chooses the minimum delay. In addition to the classical queries supported by UPPAAL, UPPAAL SMC also uses an extension of weighted metric temporal logic (WMTL) [15] to provide probability evaluation $Pr(*_{x \leq C} \phi)$, where $*$ stands for $\Diamond(eventually)$ or $\Box(always)$, which calculates the probability that ϕ is satisfied within cost $x \leq C$, but also hypothesis testing and probability comparison. In this paper, we will analyze only properties of the type "probability evaluation".

10.3 A Framework for Formal Analysis of AAL Systems: Proposed Methedology

In this section, we present in detail the framework that we propose for modeling and verification of the AAL system architectures. We consider a generic architecture category for AAL systems that supports a variety of assisted living functionalities including health monitoring, home monitoring, fall detection, user interactions, and communication with family, caregivers. Accordingly, the architecture supports a variety of components like sensors, a data collector unit to collect the sensor data, local and cloud processing, and intelligent decision support. The system architecture and its requirements are explained in detail in Section 10.4. This architecture design and the requirements in natural language form the input to our analysis framework. As depicted in Fig. 10.1, the framework is composed of the following steps:

Step 1. *Create an abstract component-based model of the proposed architecture in AADL.*

This step focuses on specifying the architecture using an architecture description language. In our case, we have chosen AADL due to its rich semantics and suitability to model real-time embedded systems. In our approach, we demonstrate the modeling of AAL systems as abstract components and show how it can be extended to suit the specific instantiations (from simpler to more complex configurations, as shown in Section 10.4). The system modeling in AADL is presented in Section 10.5.

Step 2. *Define a semantic encoding of AADL model as an NSTA model.*

Following the AADL modeling, in Step 2, we define the semantic anchoring of the AADL model as NSTA (Section 10.6). We present the semantic anchoring of the generic model and also show the the above-mentioned instantiations of the latter to various configurations of increasing complexity. The NSTA model so formulated can be further analyzed via exhaustive model checking or statistical model-checking, depending upon the technique's ability to cope with the model's complexity. For the simple architecture configuration, we use exhaustive verification with UPPAAL and for the

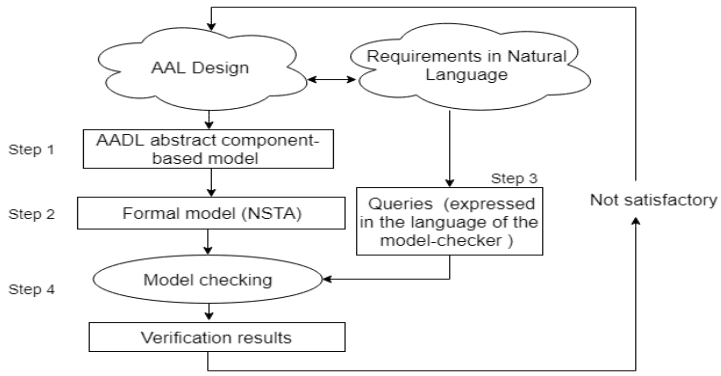


Figure 10.1: Methodology overview.

complex configuration, we use statistical model checking, using the tool UPPPAAL SMC.

In the subsequent step, the functional and non functional requirements of the architecture, which are initially specified in natural language are formalized as Timed Computation Tree Logic (TCTL) or Weighted Metric Temporal Logic(WMTL) queries to enable analysis in the NSTA model, using UPPAAL or UPPPAAL SMC. Thus, Step 3 is formulated as follows:

Step 3. *Formalize the system requirements as queries expressed in the input language of the chosen model-checker.*

As the final step, we verify the queries against the NSTA model of the architecture and gather the results (exact for UPPAAL and statistical for UPPAAL SMC) leading to Step 4 formulated as below:

Step 4. *Verify the queries in the model checker and gather verification results.*

If the verification results do not meet the requirements, we feedback information from the verification (counter example or statistical information) to our design, which we modify and iterate steps 1, 2, 3 and 4.

10.4 A Generic AAL System Architecture

In this section, we detail the generic AAL system architecture that we propose. In addition, we also present the design of a novel decision support system for our system architecture that supports the integration of multiple functionalities and provides efficient decision making by combining multiple artificial-intelligent (AI) techniques as

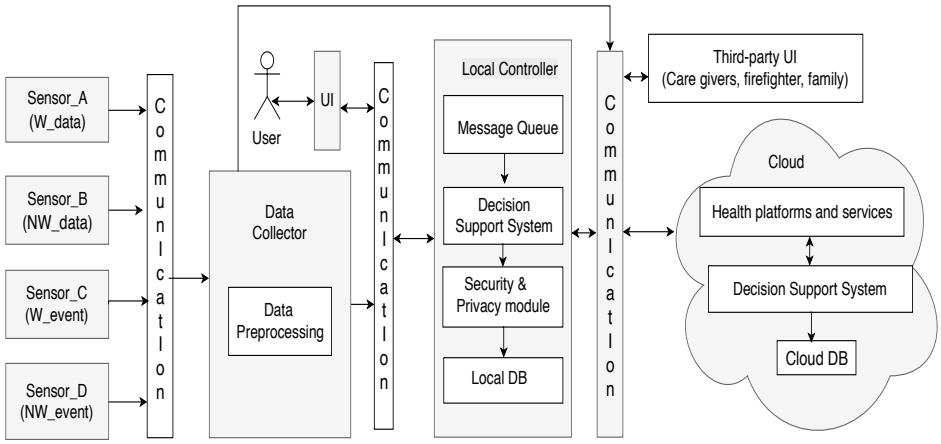


Figure 10.2: The generic AAL system architecture.

detailed later in this section. Finally, we present three specific instantiantions of the generic architecture model that follow the same modeling paradigms, yet vary in their degree of complexity with respect to integrated functionalities.

The generic AAL system architecture is presented in Fig.10.2, and follows the architecture of many commercial AAL systems with various sensors, a data collector, DSS, security and privacy, database (DB) systems, user interfaces (UI), and cloud computing support. This architecture can act as a base for the development of many integrated AAL system architectures. We classify the sensors in the AAL environment as follows:

- Wearable sensors that send information as data (W_data), e.g., sensors measuring health parameters like pulse, ECG, etc. They are represented by Sensor_A category in Fig 10.2;
- Non-wearable sensors measuring ambient parameters and health parameters (NW_data), e.g., camera sensors, motion sensors, etc., represented by Sensor_B category;
- Wearable sensors that detect events (W_event), e.g., fall sensors, marked as Sensor_C category;
- Non-wearable sensors detecting events (NW_event), e.g., fire sensors, denoted by Sensor_D category.

A particular instantiation of the generic architecture can contain n sensors of each category, respectively, $n \in N$. As depicted in Fig.10.2, the data from the sensors are collected by the Data Collector unit, which processes the data by assigning labels and

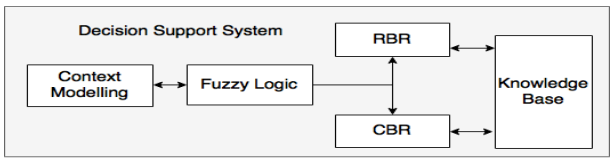


Figure 10.3: The DSS architecture

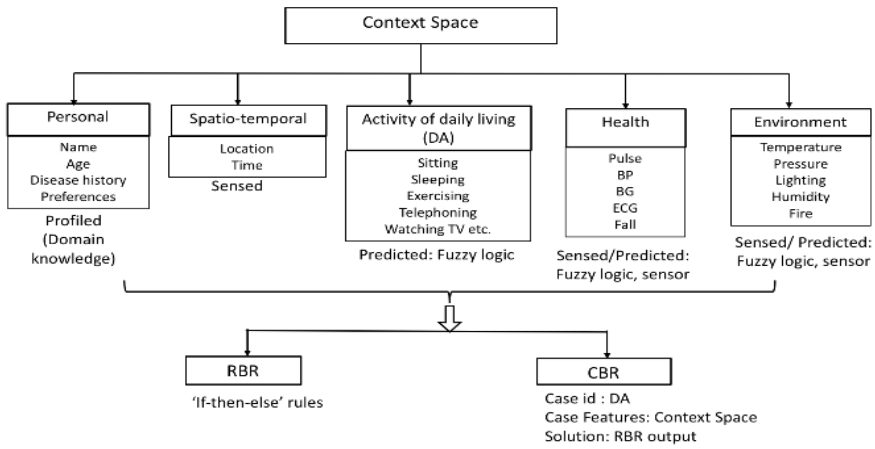


Figure 10.4: Internals of the DSS architecture (List of AI techniques)

priorities. The Data Collector sends the data to the message queue in the Local Controller, where it gets sorted according to its priority such that when the DSS processes the first element in the queue, it processes the message with the highest priority. Our architecture has both local and cloud-based processing in order to ensure fault tolerance with respect to the DSS. The components of the architecture can interact via various communication protocols.

The crux of our AAL system is the **intelligent context-aware DSS**, shown in Fig.10.3. The novelty of our architecture stems from the combination of various AI algorithms, like rule-based reasoning (RBR), fuzzy logic, and case-based reasoning(CBR) with context reasoning for efficient decision-making, as detailed below.

Our DSS architecture is inspired by the work of Zhou et al. [16], where the authors have proposed a context-aware, CBR-based ambient-intelligence system for AAL applications. CBR reasoning works very well in scenarios that are not specific and need to adapt accordingly with inputs. For instance, CBR reasoning is suited in a clinical decision support system that prescribes medicines/treatment, where the treatment, pre-

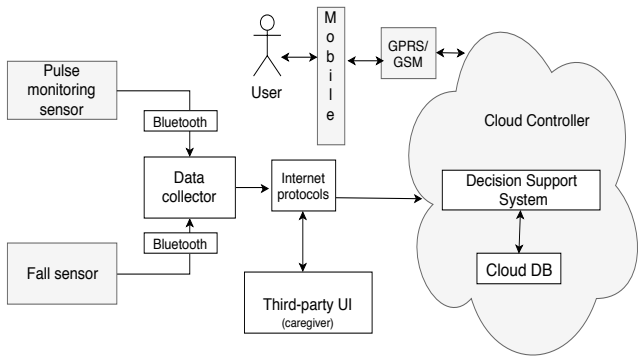


Figure 10.5: Category 1: A minimal configuration

scription and medicine dosage vary depending upon individual patients. CBR is an attractive choice due to its reasoning technique resembling more of human problem-solving competence, (i.e., trying to reason out a new scenario by looking at the similar solved cases in the past and adapting them according to the current needs) and less of knowledge engineering, however there are many scenarios that are specific and involve domain expertise, where RBR can be employed with more efficiency and ease. For instance, if a fire occurs at home, the action to be taken by the system is to notify the firefighters, which can be easily implemented using *"if-then-else"* rules rather than via a CBR system that needs to compare across all cases using a case-matching algorithm to retrieve a matching case and act accordingly. Moreover, RBR systems using fuzzy logic are very efficient to determine sensor data deviations compared to crisp logic. For instance, the normal pulse range of a person is between 60-120, and a crisp rule-based-reasoning system (Boolean logic) will classify a pulse value of 59.5 or 120.5 as an abnormal range (which in reality is not) and raises a pulse- deviation alarm to the care-giver. Using fuzzy logic, a degree of membership can be associated to each value, i.e., a pulse value of 59.5 or 120.5 is strictly not within abnormal or normal boundaries, rather it is considered 97% within normal range and 3% within abnormal range. Thus, by replacing the crisp boolean logic with fuzzy logic, a multitude of false pulse deviation alarms can be avoided. However, RBR (even fuzzy based) cannot work efficiently in many other ill-defined scenarios that require adaptability, like that of a clinical decision support system or a system that sends personalized recommendations to its users.

The DSS triggers the various AI algorithms based on a change in *context* [16]. The context-modeling (CM) and the usage of different AI algorithms are depicted in Fig. 10.4. As indicated, CM module identifies the context space based on: (i) the personal profile of the user, e.g., gender, age, disease history etc., (ii) the activity of daily living (DA) performed by the user, e.g., exercising, sleeping etc., (iii) spatio-temporal properties, like time, location of the user, etc., (iv) environmental, e.g., temperature,

pressure, fire, etc., and (v) health parameters, for instance, like blood pressure, pulse, etc. Each of these context-space components can be associated with one of the three properties - *sensed, profiled* or *predicted*. *Sensed* contexts are those directly derived from sensor values. However, predicted contexts correspond to the output resulting from further analysis of sensed inputs, e.g., activity-recognition. *Profiled* values are usually descriptive and remain unchanged.

In our DSS, fuzzy reasoning is used for detecting DA [17], and also for determining sensor-data deviations. For simplicity, we have not considered DA detection using fuzzy logic in our further modeling and analysis and has often assumed that the user's DA is known (although this is not the actual case). To take decisions in various situations, we employ RBR first, CBR as second paradigm, i.e., upon a change in context, the RBR triggers first and checks if there exists a rule to handle that particular context, if not, it allows the CBR system to tackle the context based on its learning from previous scenarios. Developing an efficient case base, case matching and formulating the adaptation rules are the most complex aspects of a CBR system. In our system, each time an RBR outputs a rule, we save it as a *case* in the CBR system with the *case-id* represented by the DA of the user, the *context space* represented by the case features, and the triggered *rule* represented by the solution for a particular case. The KB stores the context, rules, and cases. The internal structure of the DSS is represented in Fig.10.4.

The generic architecture, and its DSS can be instantiated to create a family of AAL architectures that follows the same design principles. In this paper, we present three such architectures and their DSS instantiations.

- **Category 1: A minimal configuration** - The minimum configuration architecture consists of the following modules: Two sensors (a fall sensor and a pulse monitoring sensor), a mobile phone UI, and cloud controller with third-party UI and DSS system with a minimum context-space information including the health data (pulse and fall) and DA. The simplified DSS employs only RBR with fuzzy logic as AI techniques. The minimal configuration is shown in Fig. 10.5.
- **Category 2: An intermediate configuration** - This instantiation (see Fig. 10.6) is more complex than the previous one and it contains sensors belonging to all four categories of the generic architecture (health monitoring sensors that detect pulse and blood pressure, smart home sensors that detect user movements, a wearable fall sensor and a set of physical exercise monitoring sensors), as well as a local controller with inbuilt data collection functionality, which forwards the data to the cloud controller. The cloud controller has a DSS with context modeling, fuzzy logic and RBR.
- **Category 3: A complex configuration** - In this category, we present the most complex version, the CAMI AAL architecture [2] derived from our generic model and represented in Fig. 10.7. It supports various sensors (e.g. A multitude of health and home monitoring sensors like the A&D UA-651 BLE blood pressure sensor [18], Fibaro temperature and motion sensor FGMS-001 [19], Fitbit bracelet [20], Vibby fall detection sensor [21], etc.), data collector, local

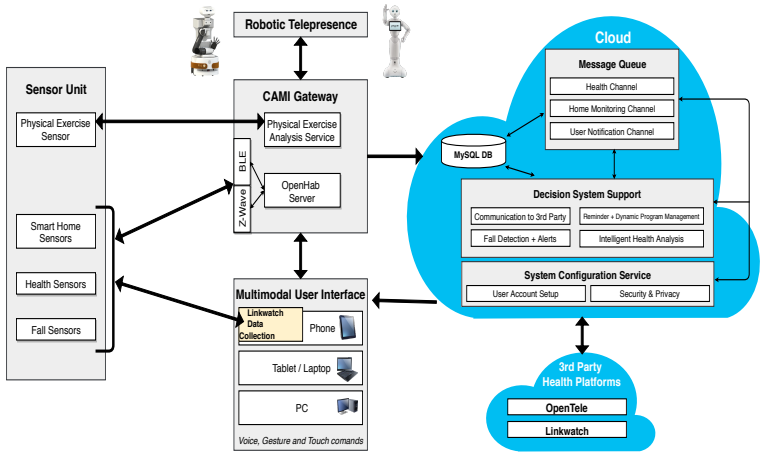


Figure 10.6: Category 2: An intermediate configuration

controller (EXYS9200-SNG [22] referred as *CAMI gateway*), the CAMI cloud, and third party health platforms like Open Tele [23] and [24]. There is a set of UI in CAMI, including robotic platforms (TIAGo [25] and Pepper [26]), mobile phone and vocal interface to facilitate the interaction with the elderly user. There is also a local backup of DSS in the CAMI gateway apart from the cloud. The communication between various modules can employ a variety of communication protocols, for instance, Bluetooth, Zigbee, Wifi, etc,. The local processor is called the CAMI gateway and is responsible for all critical functionalities. The Message Queue is implemented by Rabbit MQ Message Broker [27]. The DSS is complex and employs context modeling, fuzzy logic, RBR and CBR. There are also redundant copies of DSS in the local controller and cloud controller.

In the following, we present the modeling and analysis of the simplest architecture (Category 1), as well as of the most complex one, the CAMI architecture (Category 3). We start by describing the use-case scenarios and system requirements of the two architecture configurations, in the following section.

10.4.1 Use Case Scenarios and System Requirements

AAL systems should assist the elderly users with a variety of health, home-related functions, as well as social inclusion ones. Let us assume the following critical scenarios where we can employ systems whose architectures conform to the ones of Categories 1 and 3 described above, respectively.

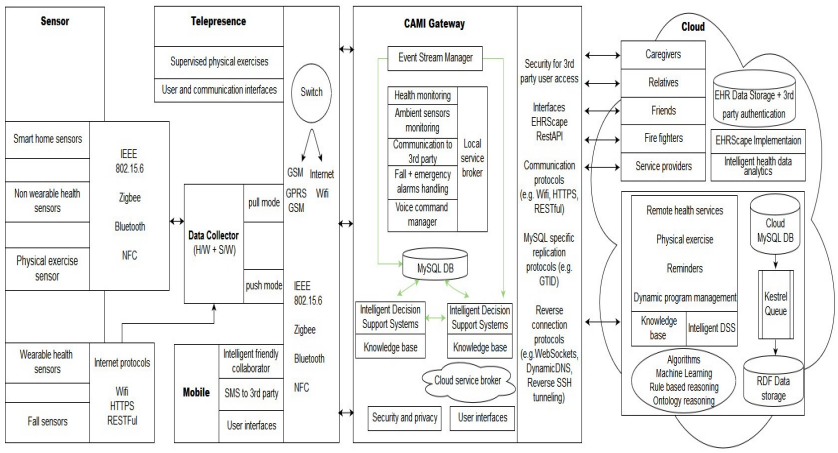


Figure 10.7: Category 3: A complex configuration: The CAMI AAL System Architecture [2]

Overall Scenario: *Jim is an elderly user living alone in his home. Jim suffers from chronic cardiac disease, slight memory loss, and falls frequently.*

If Jim uses the AAL system architecture of category 1, the latter should assist in fulfilling the scenarios below:

- *Scenario 1 - Assistance for detecting health parameter deviations:* Jim has sudden pulse variations, detected by the pulse monitoring sensor, which are critical for cardiac patients. If the pulse is low, the DSS alerts the caregiver of a low pulse. If the pulse is high and the user is currently exercising (if this is the case, a high pulse is considered as normal) and if not, it sends an alert to the caregiver.
- *Scenario 2 - Fall detection*: Jim falls heavily while exercising, the fall sensors detect the fall and the system immediately notifies the fall event to the caregiver.

However, if Jim needs additional functionality support, then he needs to acquire the CAMI AAL system (Category 3), which can handle additional scenarios besides the already mentioned ones. The fall detection in CAMI is complex, as it employs a combination of wearable fall sensor (Vibby) and camera sensor for detecting the fall event.

- *Scenario 3 - Home-monitoring functionalities*: Jim forgets to switch off the cooker after cooking his dinner, which results in a fire in the house. The fire detection sensor of CAMI detects the fire and the system alerts the firefighters of the fire incident in Jim's house.
- *Scenario 4 - Combining various functionalities in case of multiple events occurring together*: Jim is cooking his breakfast. He suddenly feels dizzy and falls.

The gas-based cooker is still on, and eventually starts a fire in Jim's house. In this case, the CAMI system detects the simultaneously occurring events, and alerts the firefighter and caregiver of both the events. As a result, the firefighters and caregivers can immediately start the rescue without waiting for alarm confirmations, avoiding potentially dangerous consequences [1]. Further, if there are any health parameter variations detected for Jim along with the fall (for instance, a low pulse), the fall event can be associated with the low pulse, and the caregiver notified accordingly, which can help in further diagnosis.

All these scenarios are safety critical and have to be processed in real time. For architecture 1, we consider verifying the following requirements:

Requirements of the minimal architecture model:

- **R1$_{\text{Arch1}}$**: If a high pulse is detected by the pulse sensor and the elderly user's DA is not exercising, then the DSS sends a notification to caregiver within 20 s. This requirement relates to Scenario 1.
- **R2$_{\text{Arch1}}$**: If a fall is detected by the fall sensor, then the DSS sends a notification to caregiver within 20 s. It is associated with Scenario 2.

Requirements of the CAMI architecture:

For the CAMI architecture, we consider verifying the following functional and quality-of-service (QoS) attributes, like fault tolerance and data consistency. Such analysis is beneficial, as the system needs to be prototyped and it offers some assessment of the system's dependability.

- **R1$_{\text{CAMI}}$**: If the fire sensor detects a fire, then the DSS sends a notification to the firefighters, within 20 s. This requirement corresponds to Scenario 3.
- **R2$_{\text{CAMI}}$**: If a fall is detected by the wearable or the camera sensor, then the DSS sends a notification to the caregiver, within 20 s. This requirement relates to Scenario 2.
- **R3$_{\text{CAMI}}$**: If there is a pulse data deviation indicating high pulse, the DA is "not exercising", and the user has a disease history of a cardiac patient, then the DSS sends a notification to the caregiver, within 20 s. This relates to Scenario 1.
- **R4$_{\text{CAMI}}$**: If fire and fall are detected simultaneously by the respective sensors, then the DSS should detect the presence of the simultaneous events and send notifications to both the firefighters and the caregiver indicating the presence of both events, within 20 s. This relates to Scenario 4.

- **R5$_{\text{CAMI}}$**: The decisions taken by the local DSS are updated in the cloud DSS such that they are eventually synchronized. This requirement relates to the data-consistency requirement of CAMI.

- **R6$_{\text{CAMI}}$**: If the local DSS fails, then the cloud DSS eventually becomes active. It corresponds to the fault-tolerance aspect of the CAMI system.

The overall goal is to analyze the satisfaction of the above requirements by the respective architectures. We achieve this by first specifying the architectures in AADL, and then by semantically mapping the specification into a (network of) STA (N(STA)) that we model check with UPPAAL (for architecture category 1) or statistically model check with UPPAAL SMC (for CAMI).

10.5 System Modelling in AADL

The generic architecture, depicted in Fig. 10.2 can be modeled in AADL as a set of interacting components. All the components are modeled as *abstract*, and can be easily extended to suit particular run-time representations appropriate for specific requirements.

In order to develop the AADL model, we classify the AADL components as:

1. **Atomic Components (AC)**: Components that do not have hierarchy in terms of sub-components with port interfaces, but might contain sub-components without port interfaces.
2. **Composite Components (CC)**: Hierarchical components that contain sub-components with and without interfaces. For example, data is a sub-component without interface and it can be part of an AC or CC hierarchy.

The system architecture itself can be considered a CC with other AC or CC as its sub-components. In order to encode the complex modeling aspects and the reasoning with functional behavior and errors, we propose a modeling format for both AC and CC as defined below.

AAL Atomic Components: An AC is defined by its component type, implementation, behaviour annex (BA), and error annex (EA). The component type definition specifies its name, category (i.e., "abstract") and interfaces. We can also specify particular component properties and flows in the type definitions[1]. The implementation of

[1]While defining the component properties, we chose to include thread-related properties like the Dispatch Protocol, Component Execution Time etc., which later aid us in reasoning. All these thread-related properties need to be instantiated by a value and hence we chose it to be instantiated with some values specific to our architecture chosen. If the reader wishes to use the AADL model

an AC defines the data sub-components. The AC's BA has two states, *Waiting* and *Operational. Waiting* represents the initial state where the component waits for an input, and *Operational* represents the state to which a component switches upon receiving the input (if it has not failed). The AC's EA uses four states to represent failure: *Failed Transient*, *LReset*, *Failed Permanent*, and *Failed ep*. The state *Failed Transient* models transient failures, from which a recovery is possible via a reset event. Since reset is modeled as an internal event that occurs with respect to a probabilistic distribution, we model an additional location *LReset* to encode a component's reset action upon the successful generation of the reset event. *Failed Permanent* models a permanent failure of the RBR, from which the component cannot recover. *Failed ep* models a failure due to error propagation from its predecessor components.

An example of an AC in the architecture is the RBR component of the CAMI DSS. In this paper, we illustrate the RBR for $R3_{CAMI}$ (Scenario 1), described in Section 10.4.1. The RBR component type, implementation, BA, and EA are shown in Listing 10.1. The component type definition specifies its name, category (i.e., "abstract") and interfaces (Lines 2-15). The RBR component type describes that it gets activated aperiodically, has an execution time of 1 s, and illustrates the data flows between the respective input and output ports. The implementation definition of RBR (Lines 16-20) defines the data sub-components like the fuzzy data output, personal information and daily activity of the user, which forms the **context-space** of Scenario 1.

In the BA (Lines 22-28), *Waiting* represents the initial state where the component waits for an input from the pulse sensor. In the *Operational* state, the system monitors the **fuzzy logic** output to identify any pulse variations. The fuzzy reasoning is not shown in Listing 10.1 as it is part of the context-reasoning module and not RBR, however we present the underlying reasoning in a nutshell. First of all, fuzzy data memberships are assigned to the range of pulse data values : Low [40 70], Normal [55 135], and High [110 300], where the numbers represent heart beats per minute. The pulse data input from the sensor are classified as Low, Normal or High. If a high pulse is detected by the RBR, then the **user context** is tracked by checking the elderly's activity of daily living and disease history. If the activity is "not exercising" and user has a cardiac disease history, a notification alert is raised and sent to the caregiver. The information is encoded as a rule in the BA depicted in Listing 10.1. Upon triggering a particular rule, the RBR output is stored in the DB as a case input for CBR, where the case-id is represented by DA, case features are the context space and the case solution is the RBR output (see Fig. 10.4). The RBR output is also synchronized with Cloud DSS such that the data consistency is maintained. In the EA (Lines 30-49), we show the states - *Waiting* and *Failed Transient*, *Failed Permanent*, *LReset* and *Failed ep* plus their transitions based on a *TF* event (event that causes transient failures), *PF* (event that causes permanent failure) and *reset*event. If a *TF* or *PF* event occurs when the component starts, the latter moves to the *Failed Transient* state or *Failed Permanent* state respectively. From *Failed*

for a specific architecture of choice, we recommend to extend the abstract models and manually update the property values under consideration or add/delete properties.

Transient, the system can generate a reset event with occurrence probability of 0.9 and moves to *LReset*. If the recovery is successful with the reset event, the system moves to *Waiting* state with probability 0.8, else it moves to *Failed Permanent* with probability 0.2. In this work, we have considered the *Waiting* state in the EA and BA to be similar. For a full description of the RBR model in AADL, the user can refer to the Appendix A.

Listing 10.1: An excerpt from the RBR component in AADL for CAMI

```
1  ——RBR (Component Type +Implementation)———
2  abstract RBR
3  features
4  input: in event data port;
5  output: out event data port;
6  flows
7  F1 : flow path input -> output;
8  properties
9  Dispatch_Protocol => Aperiodic;
10 property_eventgeneration::AperiodicEventGeneration=>1.0;
11 property eventgeneration ::Distribution=> Exponential;
12 property_failure_recovery::FailureRecoveryRate=>1.0;
13 property_failure_recovery::Distribution=> Exponential;
14 Compute_Execution_Time =>1s..1s;
15 end RBR;
16 abstract implementation RBR.impl
17 fuzzy_out_pulse:data  fuzzified_data_pulse;
18 DA: data ADL;
19 u_profile: data user;
20 end RBR.impl
21 —BA—
22 states
23 Waiting: initial complete final state;
24 Operational: state;
25 transitions
26 Waiting -[on dispatch input]->Operational
27 {if (fuzzyo_pulse=high and DA!= exercising and u_prof =cardiac_patient)
28 {output:= not_caregiver_highpulse}
29 —EA—
30 states
31 Waiting: initial state;
32 Failed_Transient: state;
33 Failed_Permanent:state;
34 LReset: state;
35 Failed_ep:state;
36 events
37 Reset: recover event;
38 TF: error event;
39 PF: error event;
40 Transitions
41 t1: Waiting -[PF]->Failed_Permanent
42 t2: Waiting -[TF]->Failed_Transient;
43 t3: Failed_Transient -[Reset]-> {LReset with 0.9,
44 Failed_Permanent with 0.1};
45 t4: LReset-[]->{Waiting with 0.8, Failed_Permanent with 0.2}
46 properties
47 EMV2::DurationDistribution => [Duration => 1s..2s; applies to Reset;
```

```
48 EMV2::OccurrenceDistribution =>[ProbabilityValue => 0.9;
49 Distribution => Fixed;] applies to Reset;
```

AAL Composite Components: A CC is defined in a similar way as that of AC, except that its BA is not explicitly defined (We assume that the behaviour of the CC is already encoded by its sub-components). Also, the EA definition of CC shows the failure behaviour of its sub-components. In Listing 10.2, we present an excerpt of the DSS component, as an example of CC. The component type definition (Lines 2-12) is similar to that of an AC, except that we do not define explicitly properties like execution time of a CC (it is considered based on the execution time of each component, respectively). However, component implementation (Lines 13-26) shows the prototypes used to define sub-components and connections between them. The EA (Lines 28-39) shows the composite error behavior of DSS and shows that the DSS moves to *Failed Transient* or *Failed Permanent*, if each of its sub-components move to these states, respectively. No BA is created for the DSS since the behavior is defined by the BA of the sub-components.

Listing 10.2: An excerpt from the DSS component in AADL for CAMI

```
1  --DSS Component Type + Implementation--
2  abstract DSS
3  features
4  input: in event data port ;
5  decision_out: out event data port;
6  properties
7  Dispatch_Protocol => Aperiodic;
8  property_eventgeneration::AperiodicEventGeneration=>10.0;
9  property eventgeneration ::Distribution=> Exponential;
10 property_failure_recovery::FailureRecoveryRate=>1.0;
11 property_failure_recovery::Distribution=> Exponential;
12 end DSS;
13 abstract implementation DSS.impl
14 prototypes
15 RBR_DSS: abstract RBR;
16 CBR_DSS: abstract CBR;
17 CM_DSS: abstract context_model;
18 subcomponents
19 RBR: abstract RBR_DSS;
20 CBR: abstract CBR_DSS;
21 CM: abstract CM_DSS;
22 connections
23 C1: port input -> CM.input;
24 C2: port CM.output-> RBR.input;
25 C3: port RBR.output-> CBR.input;
26 C4: port CBR.output-> decision_out;
27 --DSS EA--
28 annex EMV2{**
29 composite error behavior
30 [RBR.Failed_Permanent and CBR.Failed_Permanent and
31  CM.Failed_Permanent] -> Failed_Permanent;
32 [RBR.Failed_Transient and CBR.Failed_Transient and
33 CM.Failed_Transient] -> Failed_Transient;
```

```
34  [RBR.Operational or CBR.Operational or
35  CM.Operational]-> Wait;
36  EMV2::OccurrenceDistribution =>[ProbabilityValue => 10;
37  Distribution =>Exponential;] applies to  Failed_Permanent,
38  Failed_Transient, Wait;
39  end composite;**};
```

The assumptions made in the AADL model are: (i) all the system components have a reliability of 99.98%, (ii) the sensors have a periodic activation, (iii) all the system components interact via ports without any delay of communication, and (iv) the output is produced in the *Operational* state and there is no loss of information during transmission.

10.6 Semantics of AAL- Relevant AADL Components

AADL is a "semi-formal" language and in order to formally verify our AAL systems specified in AADL, we give formal semantics to AADL components (of the type used in this paper) in terms of *stochastic timed automata*, to be able to encode annex behaviors also. First, we provide the tuple definition of AADL components (Section 10.6.1), after which we perform a semantic anchoring of the AADL component tuple via a mapping between the elements of the AADL and the elements of the STA (Section 10.6.2).

10.6.1 Definition of AADL Components for AAL

An AADL component that we employ in this paper can be defined as a tuple:

$$AADL_{\text{Comp}} = \langle Comp_{\text{type}}, Comp_{\text{imp}}, EA, BA \rangle, \tag{10.3}$$

where $Comp_{\text{type}}$ represents the component type, and $Comp_{\text{imp}}$ represents the component implementation, BA the behavioral annex specification, and EA the error annex, as follows:

- $Comp_{\text{type}}$ is defined as a tuple: $Comp_{\text{type}} = \langle Features, Flow_{\text{spec}}, Prop \rangle$, where:
  - $Features = IN_{\text{p}} \cup OUT_{\text{p}}$, where IN_{p}, OUT_{p} represent the sets of $input\ ports$ and $output\ ports$ respectively, and IN_{p}, $OUT_{\text{p}} \in \{data\text{-}ports, event\text{-}ports, event\text{-}data\text{-}ports\}$;
  - $Flow_{\text{spec}} = \langle Flow_{\text{so}}, Flow_{\text{p}}, Flow_{\text{si}} \rangle$, where $Flow_{\text{so}}$, $Flow_{\text{p}}$, $Flow_{\text{si}}$ represent flow sources, flow paths and flow sinks respectively. Let $F_{\text{s0}} : Flow_{\text{so}} \rightarrow OUT_{\text{p}}$ be a function that associates certain OUT_{p} to $Flow_{\text{so}}$ with $Flow_{\text{so}} \subseteq OUT_{\text{p}}$, $F_{\text{p}} : Flow_{\text{p}} \rightarrow OUT_{\text{p}} \times IN_{\text{p}}$ be a function that associates and an input and an output to a flow, and $F_{\text{si}} : Flow_{\text{si}} \rightarrow IN_{\text{p}}$

be a function that associates certain IN_{p} to $Flow_{\text{si}}$, with $Flow_{\text{si}} \subseteq IN_{\text{p}}$. For instance, in our AAL architecture, we can define $Flow_{\text{spec}}$ for fall events by defining the output port of the fall sensor as $Flow_{\text{so}}$, the input port of the cloud DSS as $Flow_{\text{si}}$, and the input and output ports of all the intermediate components defining the Flow_{p};

- $Prop$ is the set of associated properties of the component, like $Deployment$, $Communication$, $Timing$, $Thread\text{-}related\ properties$, $user\text{-}\ defined\ properties$, etc. [11]. In this work, we only consider a subset of $Timing$, $Thread\text{-}related\ properties$ and $user\text{-}\ defined\ properties$ that are represented as follows: $Prop = \{T_{\text{p}}, T_{\text{e}}, Dispatch\ protocol, Userprop\}$ where T_{p} and T_{e} represent the period and execution-time of the component, respectively, T_{p}, $T_{\text{e}} \in Timing\ properties$, $Dispatch\ protocol \in \{P, AP\}$[2], where P represents a Periodic and AP represents an Aperiodic protocol, and $P, AP \in Thread\text{-}related\ properties$, and $Userprop \in \{event\_gen\_dist, failure\_recovery\_dist\}$ defines the set of user-defined properties used for specifying the occurrence distribution of aperiodic events ($event\_gen\_dist$) and failure recovery ($failure\_recovery\_dist$).

- $Comp_{\text{imp}}$ is defined as $Comp_{\text{imp}} = \langle SC, P_{\text{t}}, Con, MSM, Flow_{\text{imp}}, ETF \rangle$, where:
  - SC represents the set of sub-components of the system with port interfaces (SC_{i}) and without port interfaces (SC_{Data}), i.e., $SC = SC_{\text{Data}} \cup SC_{\text{i}}$;
  - P_{t} denotes the set of *Prototypes* used to define SC via $Fp : P_{\text{t}} \rightarrow SC_{\text{i}} \times SC_{\text{Data}}$, a function that associates SC to a P_{t}, respectively;
  - Con represents the set of connections. $F_{\text{con}} : Con \rightarrow Features$ is a function that assigns $Features$ to Con;
  - MSM is the mode state machine that is modeled by a tuple, as follows: $MSM = \langle M_{\text{s}}, \rightarrow \rangle$, where M_{s} is the set of states, and $\rightarrow \subseteq M_{\text{s}} \times ev \times M_{\text{s}}$ is the transition relation (with ev being the set of events, such that $Fe : event\text{-}ports \rightarrow ev$, $event\text{-}ports \in Features$). We write $s \xrightarrow{e} s'$ as short for $(s, e, s') \in \rightarrow$, where $s, s' \in Ms$, and $e \in ev$. The set of Con is defined with respect to MSM, if present;
  - $Flow_{\text{imp}}$ are the flow implementations, represented as $Flow_{\text{imp}} : SC \rightarrow Flow_{\text{spec}}$;

[2]The dispatch protocol property of a thread determines when the thread is executed. A periodic thread is activated at time intervals of the specified period T; an aperiodic thread is activated when an event arrives at a port of the thread.

  - ETF represents the set of end-to-end flows as complete flow paths from a starting SC_{i} to the final SC_{i}, respectively.

- The error annex EA is defined as the tuple: $EA = \langle E_{\text{flows}}, E_{\text{beh}}, E_{\text{prop}} \rangle$, where:
  - E_{flows} denotes the error flows, $E_{\text{flows}} = \langle E_{\text{pp}}, Err_{\text{so}}, Err_{\text{p}}, Err_{\text{si}} \rangle$, where E_{pp} describes error propagations, and Err_{so}, Err_{p}, Err_{si} represents error sources, error paths, and error sinks, respectively; $F_{\text{e1}} : Err_{\text{so}} \rightarrow OUT_{\text{p}}$ is a function that associates certain output ports with error sources, $F_{\text{e2}} : Err_{\text{p}} \rightarrow IN_{\text{p}} \times OUT_{\text{p}}$ is a function that associates input and output ports via Err_{p}, $F_{\text{e3}} : Err_{\text{si}} \rightarrow IN_{\text{p}}$ is a function that assigns certain input ports as error sinks;
  - E_{beh} represents error behavior, $E_{\text{beh}} = \langle E_{\text{s}}, \rightarrow_{\text{e}}, E_{\text{e}}, EM_{\text{Comp}} \rangle$, where E_{s} represents the set of error states, $\rightarrow_{\text{e}}$ denotes an error transition relation, $\rightarrow_{\text{e}} \subseteq E_{\text{s}} \times E_{\text{e}} \times E_{\text{s}}$, with E_{e}, the set of error events. For a CC, the error behavior is represented as EM_{Comp} (error-model for a CC) with respect to the failure of its SC_{i}. Let s_{e} and $s_{\text{e}}{}'$ be two error states, $s_{\text{e}}, s_{\text{e}}{}' \in E_{\text{s}}$, and $\rightarrow_{\text{e}}$ the transition between them due to an error event $e_{\text{e}} \in E_{\text{e}}$, then $s_{\text{e}} \xrightarrow{e_{\text{e}}}_{\text{e}} s_{\text{e}}{}'$. We represent initial state as $s_{0\text{e}} \in E_{\text{s}}$. $F_{E_{\text{pp}}} : Epp \rightarrow IN_{\text{p}} \times OUT_{\text{p}}$ is a function that associates input and output ports to error propagations;
  - E_{prop} denotes the error properties. In our work, we focus only on two error properties: *Duration distribution* (Dur_{dist}), and *Occurrence distribution* ($Occur_{\text{dist}}$), which aid in our error analysis, thus $E_{\text{prop}} = \{Dur_{\text{dist}}, Occur_{\text{dist}}\}$.
- The Behaviour Annex, BA is defined as: $BA = \langle B_{\text{v}}, B_{\text{s}}, \rightarrow_{\text{b}} \rangle$, where B_{v}, B_{s}, represent the set of variables, and the states of BA, respectively and $\rightarrow_{\text{b}}$ is a BA transition relation. Let s_{b} and s'_{b} be two states of BA, $s_{\text{b}}, s'_{\text{b}} \in B_{\text{s}}$, and $\rightarrow_{\text{b}}$ the transition between them, $\rightarrow_{\text{b}} \subseteq B_{\text{s}} \times B_{\text{v}} \times SC_{\text{Data}} \times B_{\text{s}}$, with SC_{Data} being the set of data subcomponents. We denote by $s_{0\text{b}} \in B_{\text{s}}$ the initial state of a BA path.

Formally, we distinguish the Atomic Component from the Composite Component as follows:

- $AC \in AADL_{\text{Comp}}$, where $Comp_{\text{ImplAC}} = \{SC_{\text{Data}}\}$, $EA_{\text{AC}} \neq \emptyset$, where $E_{\text{beh}} \in EA_{\text{AC}} = \{E_{\text{s}}, \rightarrow_{\text{e}}, E_{\text{e}}\}$, $BA_{\text{AC}} \neq \emptyset$,
- $CC \in AADL_{\text{Comp}}$, where $Comp_{\text{ImplCC}} = \{P_{\text{t}}, SC_{\text{i}}, SC_{\text{Data}}, Con, MSM, Flow_{\text{imp}}, ETF\}$, $EA_{\text{CC}} \neq \emptyset$, where $E_{\text{beh}} \in EA_{\text{CC}} = \{EM_{\text{Comp}}\}$, $BA_{\text{CC}} = \emptyset$. A CC represents the system-level view of the architecture.

Next, we present an instantiated example of an AC and a CC from the CAMI architecture. The RBR component of DSS is an AC and it is defined by its type, implementation, BA, and EA (Listing 10.1). In formal semantics, we define it as follows:

$$RBR_{\text{AADL}} = \langle Comp_{\text{type RBR}}, Comp_{\text{imp RBR}}, EA_{\text{RBR}}, BA_{\text{RBR}}, \rangle \tag{10.4}$$

where the elements are defined as follows:

- $Comp_{\text{type RBR}} = \langle Features_{\text{RBR}}, Flow_{\text{spec RBR}}, Prop_{\text{RBR}} \rangle$, with:
  - $Features_{\text{RBR}} = IN_{\text{p}} \cup OUT_{\text{p}}$, and $IN_{\text{p}}, OUT_{\text{p}} \in \{ event\text{-}data\text{-}ports \}$,
  - $Flow_{\text{spec RBR}} = \langle Flow_{\text{p}} \rangle$,
  - $Prop_{\text{RBR}} = \{T_{\text{e}}, AP, event\_gen\_dist, failure\_recovery\_dist\}$.
- $Comp_{\text{imp RBR}} = \langle SC_{\text{DataRBR}} \rangle$
- $EA_{\text{RBR}} = \{E_{\text{pp}}, Err_{\text{p}}, E_{\text{s}}, \rightarrow_{\text{e}}, E_{\text{e}}, Dur_{\text{dist}}, Occur_{\text{dist}}\}$
- $BA_{\text{RBR}} = \{B_{\text{s}}, \rightarrow_{\text{b}}\}$

On the other hand, the DSS in our CAMI architecture is a CC, with multiple sub-components and hence it is defined by its type, implementation and EA (no BA) as shown in Listing 10.2. Formally, it can be represented as follows:

$$DSS_{\text{AADL}} = \langle Comp_{\text{type DSS}}, Comp_{\text{imp DSS}}, EA_{\text{DSS}} \rangle \tag{10.5}$$

where the elements are defined as follows:

- $Comp_{\text{type DSS}} = \{Features_{\text{DSS}}, Flow_{\text{spec DSS}}, Prop_{\text{DSS}}\}$, where:
  - $Features_{\text{DSS}} = IN_{\text{p}} \cup OUT_{\text{p}}$, and $IN_{\text{p}}, OUT_{\text{p}} \in \{event\text{-}data\text{-}ports\}$,
  - $Flow_{\text{spec DSS}} = \langle Flow_{\text{p}} \rangle$,
  - $Prop_{\text{DSS}} = \{AP, event\_gen\_dist, failure\_recovery\_dist\}$.
- $Comp_{\text{imp DSS}} = \{SC_{\text{DSS}}, Pt_{\text{DSS}}, Con_{\text{DSS}}, Flow_{\text{imp DSS}}\}$, where:
  - $SC_{\text{DSS}} = \{CM, RBR, CBR\}$,
  - $Pt_{\text{DSS}} = \{CM, RBR, CBR\}$,
  - $Con_{\text{DSS}} = \{IN_{\text{pDSS}} \rightarrow IN_{\text{pCM}}, OUT_{\text{pCM}} \rightarrow IN_{\text{pRBR}}, OUT_{\text{pRBR}} \rightarrow IN_{\text{pCBR}}, OUT_{\text{pCBR}} \rightarrow OUT_{\text{pDSS}}\}$,
  - $Flow_{\text{imp DSS}} = \{CM \rightarrow Flow_{\text{p}}, RBR \rightarrow Flow_{\text{p}}, CBR \rightarrow Flow_{\text{p}}\}$.
- $EA_{\text{DSS}} = \{EM_{\text{Comp}}\}$

In the next sub-section, we present our semantic encoding of atomic and composite components, in terms of NSTA.

10.6.2 Formal Encoding of AADL Components as NSTA

Using the definition of AADL components given in Section 10.6.1, the formal definition of STA as $STA = \langle L, l_0, A, V, C, E, I, \mu, \gamma \rangle$, and of $NSTA = ||_i STA_i$ (see Section 10.2.2), we define a semantic encoding of the AADL components, respectively, in terms of NSTA.

Definition 1 (Formal Encoding of AC)**.** *Any atomic component in AADL, defined by:* $AC = \langle Comp_{typeAC}, Comp_{implAC}, EA_{AC}, BA_{AC} \rangle$ *is encoded as an NSTA as follows:* $AC \rightsquigarrow NSTA_{AC} = AC_{iSTA} || AC_{aSTA}$*, where* AC_{iSTA} *is the so-called "Interface STA" of AC, which corresponds to* $Comp_{typeAC}$ *and* $Comp_{implAC}$*, whereas* AC_{aSTA} *is the "Behavioral STA" that encodes the EA and BA of an AC.*

- *The* $\mathbf{AC_{iSTA}}$ *is defined according to a template STA (see Fig. 10.8) with* $L \in \{Idle, Op, Fail, start, stop\}$*,* $l_0 = Idle$*, Op corresponds to the Operational state of the RBR, start, stop represent the locations to initiate the synchronizations with* AC_{aSTA} *and* $E = \{Idle \longrightarrow start, start \longrightarrow Op, Op \longrightarrow stop, stop \longrightarrow Idle, Op \longrightarrow Fail, Fail \longrightarrow Idle\}$*. This template is annotated with the following information:*
  - $V = out\_port \cup in\_port \cup \{PF, TF\} \cup SC_{Data}$ *, where out_port and in_port represent the set of output and input ports* $\in \{data\text{-}ports, event\text{-}ports, event\text{-}data\text{-}ports\}$*, respectively, and the Boolean variables,* PF, TF*, represent the error events associated with the transient failure and permanent failure of AC, plus the variable associated with* $SC_{Data} \in Comp\_imp$*;*
  - $C = \{x\}$ *is the set of clocks that models the period and execution time of AC;*
  - $A = \{start\_ACi?, start\_AC!, stop\_AC!, stop\_ACi!\} \cup \{x = 0\}$*, where A is the set of synchronization channels associated with input-output ports* $\in \{event\text{-}data\text{-}ports, event\text{-}ports\}$*, that is, channels* $start\_AC!$*,* $stop\_AC!$*, and the synchronization channels for the interface of the corresponding CC, that is,* $start\_ACi?, stop\_ACi!$ *and the reset actions on* x*;*
  - $E = \{Idle \xrightarrow{start\_ACi? \wedge x == T_p} start, start \xrightarrow{start\_AC!, x=0} Op,$ $Op \xrightarrow{TF\_AC==1 \vee PF\_AC==1} Fail, Op \xrightarrow{x==T_e, stop\_AC!} stop,$ $stop \xrightarrow{stop\_ACi!} Idle, Fail \xrightarrow{TF\_AC==0 \wedge PF\_AC==0} Idle,$ $Fail \xrightarrow{TF\_AC==1 \wedge PF\_AC==1} Fail\}$*, where E is defined by the template populated with A and guards that ensure the correctness of transitions.*
  - $I(Idle) = (x \leq T_p)$*, if the dispatch protocol associated with AC is periodic, and* $I(Op) = (x \leq T_e)$*, where* T_p *and* T_e *represent the period and execution-time of AC;*

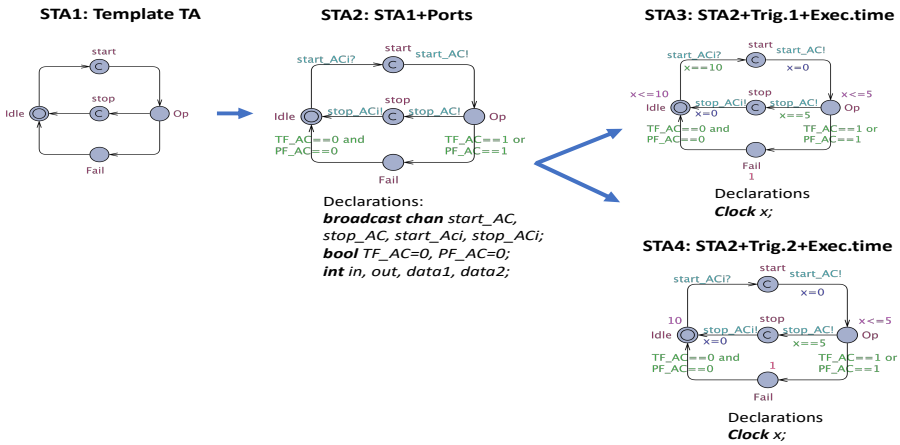


Figure 10.8: Step-by-step formulation of AC_{iSTA}

- $P(Idle) = \mu_1$, *and* $P(Fail) = \mu_2$, *where* $P(Idle) = \mu_1$ *represents the occurrence distribution of aperiodic event (if the dispatch protocol associated with AC is aperiodic), and* $P(Fail) = \mu_2$ *represents the probability of leaving location* $Fail$*;*

- *The* $\boldsymbol{AC_{aSTA}}$ *is created in a similar way with:*
  - $L = \{Wait, Op, TrF, PrF, Fail\_ep, LReset, L1, L2\}, l0 = Wait$, *where* L *comprises the set of states in EA and BA (Wait, Operational (Op), Transient Failure (TrF), Permanent Failure (PrF), Failed due to error propagation (Fail_ep), and reset location (LReset), plus additional committed locations* $(L1, L2)$ *that ensure that receiving is deterministic in UPPAAL SMC;*
  - $A = \{start\_AC?, stop\_AC?\} \cup \{action_{BA,EA}(), TF = 0, TF\_AC = 1,\ PF\_AC = 1, reset\_AC = 0, reset\_AC = 1, err\_pAC = 0, err\_pAC = 1, err\_p = 1, y = 0\}$, *where* A *is composed of the actions defined in BA and EA (*$action_{BA,EA()}$*), plus the synchronizations channels to concord with* AC_{iSTA} *(*$start\_AC?, stop\_AC?$*), and the reset of clock* y*;*
  - $V = \{PF\_AC, TF\_AC, reset\_AC, err\_pAC\}$, *where* V *consists of the set of error events defined in the EA, that is, PF_AC : Permanent Failure of AC, TF_AC: Transient Failure of AC, reset_AC: Reset of AC, err_pAC: error propagation of AC;*

- $C = \{y\}$ *is the clock that measures the time elapsed for reset action of a particular component;*
- $E = \{Wait \xrightarrow{start\_AC?} L1, L1 \xrightarrow{TF\_AC=1,err\_pAC=1} TrF,$
  $L1 \xrightarrow{PF\_AC=1,err\_pAC=1} PrF, L1 \rightarrow L2, L2 \rightarrow Op,$
  $Op \xrightarrow{stop\_AC?,action_{BA,\ EA}()} Wait, TrF \xrightarrow{reset\_AC=1,y=0} LReset,$
  $TrF \xrightarrow{PF\_AC=1,err\_pAC=1,reset\_AC=0} PrF,$
  $LReset \xrightarrow{TF\_RBR=0,err\_pAC=0,reset\_AC=0} Wait,$
  $LReset \xrightarrow{PF\_AC=1,err\_pAC=1,reset\_AC=0} PrF,$
  $Wait \xrightarrow{err\_p==1} Fail\_ep\}$, *where E consists of the transitions in EA, BA and those between* $L1$ *and* $L2$*;*
- $I(LReset) = (y \leq Dur_{dist(Reset)})$*;*
- $P(Wait) = \mu$*, that is the occurrence-distribution of* $Wait$*;*
- $L1 \xrightarrow{\gamma_1} L2$, $L1 \xrightarrow{\gamma_2} TrF$, $L1 \xrightarrow{\gamma_3} PrF$, *where* $\gamma_1, \gamma_2, \gamma_3$, *are defined according to the occurrence-distribution of the error events.* □

Definition 2 (Formal Encoding of CC)**.** *The formal encoding of a CC defined by the tuple:* $CC = \langle Comp_{typeCC}, Comp_{implCC}, EA_{CC} \rangle$ *is also a network of two synchronized STA,* $CC_{NSTA} = CC_{iSTA} || CC_{aSTA}$*, where* CC_{iSTA} *is the "interface" STA of the CC component, and* CC_{aSTA} *is the "annex" STA that encodes the information from the error annex in AADL.*

- *The* $\boldsymbol{CC_{iSTA}}$ *is defined by formally encoding* $(Comp_{typeCC}, Comp_{implCC})$*, as follows:*
  - $L = \{Wait, Fail\} \bigcup_{i=1}^{n} \{L_i Sync\} \bigcup_{i=1}^{n} \{SC_i\}$*, where L contains one location for each sub-component defined by SC, one additional location for each sub-component that ensures the correct synchronization, location* $Fail$ *to model the component failure, and* $Wait$ *to model the initial location;*
  - E *is defined according to* Con*. For each connection in* Con*, we define 2 edges,* $l \longrightarrow L_i Sync$ *and* $L_i Sync \longrightarrow l'$*, where l, l'*$\in L$ *are locations created based on the sub-components for which the connections are defined, and* $L_i Sync \in L$ *is a location created for synchronization;*
  - $V = out\_port \cup in\_port \cup \{PF, TF\} \cup SC_{Data}$ *, where out_port and* in_port *represent the set of output and input port variables* $\in \{data\text{-}ports, event\text{-}ports, event\text{-}data\text{-}ports\}$*, respectively, and the Boolean variables,* PF, TF*, represent the error events associated with the transient failure and permanent failure of CC, plus the variable associated with* $SC_{Data} \in Comp\_imp$*;*

- $C = \{x\}$ *if* $T_p \neq \emptyset$*;*
- *A is defined based on the updates defined by* MSM*, the updates defined by* $Flow_{imp}$*, the synchronizations defined by* Con*, the synchronization with* CC_{aSTA}*,* AC_{aSTA}*, and in case* C *is not void, we add the clock reset of the clock(s) in C;*
- $I(Wait)$=$(x \leq T_p)$ *if* $T_p \neq \emptyset$*;*
- $P(l) = \mu$*, where* $l \in L$ *and* μ *is defined by* $Prop$*.*

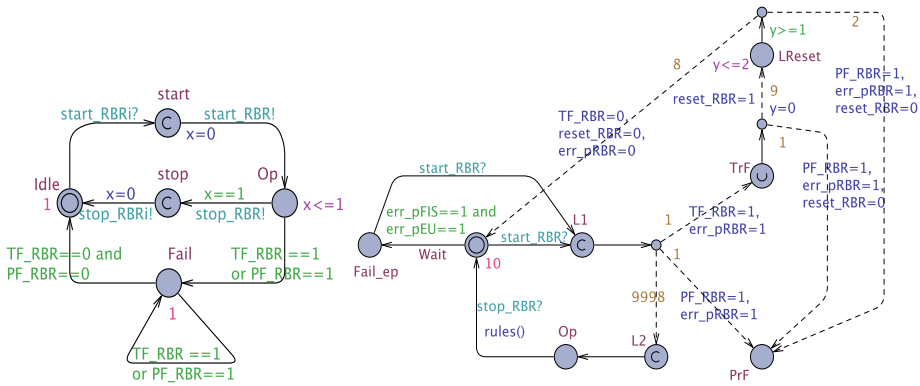
- ***CC**$_{aSTA}$ is defined as follows:*
  - $L = E_s \in EA$*,* $l_0 = s_{0e} \in E_s$*, where* E_s *is the set of states of EA;*
  - $E = \rightarrow_e$*;*
  - $A = \{TF\_CC = 1, TF\_CC = 0, PF\_CC = 1\}$*;*
  - V *is represented by the global variables defined in* CC_{iSTA}*;*
  - $C = \emptyset$*;*
  - $P(l) = \mu$*, where* $l \in L$ *and* μ *is defined by* $Occur_{dist} \in E_{prop}$*.*

All the other CC elements are transformed based on the encoding EA of AC. □

Next, we show the rules instantiated on our previously selected AADL components of CAMI, that is, RBR and DSS, as examples of transforming AC and CC into corresponding STA. There are also some additional transitions defined which are not the direct result of applying the rules, but are needed due to the requirements of our modeling tool, UPPAAL SMC.

The RBR_{AADL} defined by Eq.(10.4), is mapped into an NSTA (RBR_{NSTA}) following the Definition 1: RBR_{NSTA}=$RBR_{\text{iSTA}}||RBR_{\text{aSTA}}$ (Fig. 10.9), where RBR_{iSTA} is the so-called "Interface STA" of RBR which corresponds to $Comp_{\text{type RBR}}$ and $Comp_{\text{impl RBR}}$, whereas RBR_{aSTA} is the "Annex STA" of RBR that encodes its EA and BA.

- The RBR_{iSTA} is formally represented as a tuple, where:
  - $L = \{Idle, Start, Op, Fail\}, l0 = \{Idle\}$
  - $A = \{start\_RBRi?, start\_RBR!, stop\_RBR?\} \cup \{x = 1\}$
  - $V = \{out\_port, in\_port, PF\_RBR, TF\_RBR\}$
  - $C = \{x\}$
  - $E = \{Idle \xrightarrow{start\_RBRi?} start, start \xrightarrow{start\_RBR!, x=0} Op,$ $Op \xrightarrow{TF\_RBR==1 \vee PF\_RBR==1} Fail, Op \xrightarrow{x==1, stop\_RBR!} Idle, Fail$ $\xrightarrow{TF\_RBR==0 \wedge PF\_RBR==0} Idle, Fail \xrightarrow{TF\_RBR==1 \wedge PF\_RBR==1}$ $Fail\}$

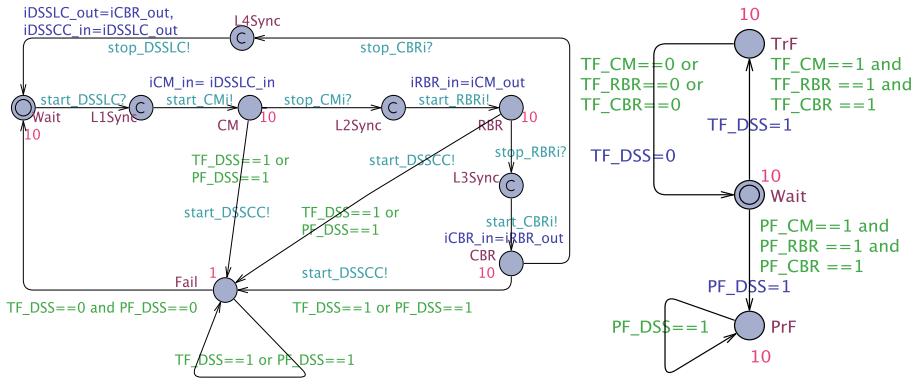


(a) Interface STA (RBR_{iSTA}) (b) Annex STA (RBR_{aSTA})

Figure 10.9: The STA for the RBR

- $I(Op)$=$(x \leq 1)$
- $P(Idle) = 1$, $P(Fail) = 1$, given by μ

- RBR_{aSTA} is defined in a similar way:
  - $L = \{Wait, Op, TrF, PrF, Fail\_ep, LReset, L1, L2, LSync\}, \{l0 = Wait\}$
  - A=$\{start\_RBR?, stop\_RBR?, stop\_RBRi!\}$ $\cup\{rules(), TF\_RBR = \{0, 1\}$
  - $V = \{PF\_RBR, TF\_RBR, reset\_RBR, err\_pRBR, err_p\}$
  - $C = \{y\}$
  - E= $\{Wait \xrightarrow{start\_RBR?} L1$, $L1 \xrightarrow{TF\_RBR=1, err\_pRBR=1} TrF$, $L1 \xrightarrow{PF\_RBR=1, err\_pRBR=1} PrF$, $L1 \rightarrow L2$, $L2 \rightarrow Op$, $Op \xrightarrow{stop\_RBR?, rules()} Lsync$, $Lsync \xrightarrow{stop\_RBRi!} Wait$, $TrF \xrightarrow{reset\_RBR=1, y=0} LReset$, $TrF \xrightarrow{PF\_RBR=1, err\_pRBR=1, reset\_RBR=0} PrF$, $LReset \xrightarrow{TF\_RBR=0, err\_pRBR=0, reset\_RBR=0} Wait$, $LReset \xrightarrow{PF\_RBR=1, err\_pRBR=1, reset\_RBR=0} PrF$, $Wait \xrightarrow{err\_p==1} Fail\_ep\}$
  - $I(LReset) = (y \leq 2)$
  - $P(Wait) = 10$, given by μ



(a) Interface STA (DSS_{iSTA}) (b) Annex STA (DSS_{aSTA})

Figure 10.10: The STA for the DSS

– $L1 \xrightarrow{0.9998} L2$, $L1 \xrightarrow{0.001} TrF$, $L1 \xrightarrow{0.001} PrF$, assigned by γ

Similarly, the DSS_{AADL}, shown in Listing 10.2, and represented by Eq.(10.5), is mapped into an NSTA: $DSS_{\text{AADL}} \rightsquigarrow DSS_{\text{NSTA}}$=$DSS_{\text{iSTA}}||DSS_{\text{aSTA}}$ (Fig. 10.10), where DSS_{iSTA} is the so-called "Interface STA" of DSS, which corresponds to $Comp_{\text{type DSS}}$ and $Comp_{\text{impl DSS}}$, whereas DSS_{aSTA} is the "Annex STA" that encodes the EA of CC.

- The tuple elements of DSS_{iSTA} are as follows:
  - $L = \{Wait, CM, RBR, CBR, Fail, L1Sync, L2Sync, L3Sync, L4Sync\}$, $l0 = \{Wait\}$
  - $A = \{start\_DSSLC, start\_CMi!, stop\_CMi?, start\_RBRi!, stop\_RBRi?, start\_CBRi!, stop\_CBRi?, stop\_DSSLC!, start\_DSSCC!\} \cup \{iCM\_in = iDSSLC\_in, iRBR\_in = iCM\_out, iCBR\_in = iRBR\_out, iDSSLC\_out = iCBR\_out, iDSSCC\_in = iDSSLC\_out\}$
  - $V = \{iDSSLC\_in, iCM\_in, iRBR\_in, iCBR\_in, iDSSCC\_in, iDSSLC\_out, iCM\_out, iRBR\_out, iCBR\_out, iDSSLC\_out, PF\_DSS, TF\_DSS\}$
  - $E = \{Wait \xrightarrow{start\_DSSLC?} L1Sync, L1Sync \xrightarrow{start\_CMi!, iCM\_in=iDSSLC\_in} CM, CM \xrightarrow{stop\_CMi?} L2Sync, L2Sync \xrightarrow{start\_RBRi!, iRBR\_in=iCM\_out} RBR, RBR \xrightarrow{stop\_RBRi?} L3Sync, L3Sync \xrightarrow{start\_CBRi!, iCBR\_in=iRBR\_out} CBR, CBR \xrightarrow{stop\_CBRi?} L4Sync, L4Sync \xrightarrow{stop\_DSSLC!, iDSSLC\_out=iCBR\_out, iDSSCC\_in=iDSSLC\_out}$

$Wait, CM \xrightarrow{(TF\_DSS=1 \vee PF\_DSS=1), start\_DSSCC!} Fail, RBR$
$\xrightarrow{(TF\_DSS=1 \vee PF\_DSS=1), start\_DSSCC!} Fail, CBR$
$\xrightarrow{(TF\_DSS=1 \vee PF\_DSS=1), start\_DSSCC!} Fail, Fail$
$\xrightarrow{(TF\_DSS==1 \vee PF\_DSS==1)} Fail, Fail \xrightarrow{(TF\_DSS==0 \wedge PF\_DSS==0)}$
$Wait\}$

- $P(Wait)$=10, $P(CM)$=10, $P(RBR)$=10, $P(CBR)$=10, $P(Fail)$=1

$EA_{\text{CC}} \rightsquigarrow DSS_{\text{aSTA}}$

- DSS_{aSTA} has the following syntactic elements:
  - $L = \{Wait, TrF, PrF\}, l0 = \{Wait\}$
  - $A = \{TF\_DSS = \{0, 1\}, PF\_DSS = \{1\}\}$
  - $V = \{TF\_DSS, TF\_CM, TF\_RBR, TF\_CBR, PF\_CM, PF\_RBR, PF\_CBR, PF\_DSS\}$
  - $E = \{Wait \xrightarrow{TF\_CM==1 \wedge TF\_RBR==1 \wedge TF\_CBR==1, TF\_DSS=1} TrF,$
    $Wait \xrightarrow{PF\_CM==1 \wedge PF\_RBR==1 \wedge PF\_CBR==1, PF\_DSS=1} PrF, PrF$
    $\xrightarrow{PF\_DSS==1} PrF, TrF \xrightarrow{TF\_CM==0 \vee TF\_RBR==0 \vee TF\_CBR==0, TF\_DSS=0}$
    $Wait\}$
  - $P(Wait) = 10, P(TrF) = 10, P(PrF) = 10$

In addition to the above description, for the reader to have a deeper understanding of modeling the AI algorithms in the respective STA, we show an excerpt of the variable declarations and functions encoding that we have used to describe our DSS AI algorithms in Listing 10.3. We show the context modeling, fuzzy reasoning and RBR in the following and also show how the successful RBR outputs are stored as cases for CBR.

In the *context modeling*, we describe our data structures that we have defined for specifying user profile, spatio-temporal properties, activity of daily living of the user, health and ambient data. The context information changes based on the sensed data and events. In the *fuzzy reasoning* module, we show how the pulse data of the user is fuzzified into low, normal and high values and the corresponding update of the context information. The *RBR* takes the input from the context modeling module and is represented by various if-then-else rules as shown. We also demonstrate how the RBR output is stored as a case in the case-base of the *CBR* module.

Listing 10.3: DSS model in STA in detail

```
——Context modeling——
typedef struct{
int user_name; //1 Jim
int age; //Age =65 years
int disease_history; //3-Heart disease
```

```
}user_profile;
user_profile up;
typedef struct{
int position;
//1= inside home, 0 -outiside home
}stemporal_properties;
typedef struct{
int pulse;
int fall_w;
int fall_c;
}health_parameters;
typedef int uADL;user_profile profile;
uADL ADL; //2-exercising, 1-resting
stemporal_properties s_temp;
health_parameters health;
ambient_parameters ambient;
typedef struct{
user_profile profile;
uADL ADL;
stemporal_properties s_temp;
health_parameters health;
ambient_parameters ambient;
}context_model;
--Fuzzy Logic Reasoning--
void fuzzify()
{
if(iFIS_in.data_val>=55 and iFIS_in.data_val<=135)
{FIS_out.health.pulse=2; }
else if(iFIS_in.data_val>=40 and iFIS_in.data_val<=70)
{FIS_out.health.pulse=1;}
else if (iFIS_in.data_val<=300 and iFIS_in.data_val>=110)
{FIS_out.health.pulse=3;}
FIS_out.health.pulse=fuzzyout_pulse;
FIS_out.profile.user_name=upro.profile.user_name;
FIS_out.profile.age=upro.profile.age;
FIS_out.profile.disease_history=upro.profile.disease_history;
FIS_out.ADL =upro.ADL;
FIS_out.s_temp.position=upro.s_temp.position;
}
void update_contextEU()
{
 FIS_outsave.ambient.fire= EU_out.ambient.fire;
 FIS_outsave.health.fall_c= EU_out.health.fall_c;
 FIS_outsave.health.fall_w= EU_out.health.fall_w;
}
--RBR --
void rules()
if ((iRBR_in.health.fall_w==1  or iRBR_in.health.fall_c==1)
and iRBR_in.ambient.fire==1)
{rule.notifications_caregiver=2;
rule.notifications_firefighter=2;}
else if (iRBR_in.health.pulse ==3 and iRBR_in.ADL==1
and iRBR_in.profile.disease_history==3)
{rule.notifications_caregiver=1;}

else if (iRBR_in.health.pulse==1 and iRBR_in.ADL==1
and iRBR_in.profile.disease_history==3)
{rule.notifications_caregiver=3;
```

```
}
else if (iRBR_in.ambient.fire==1)
{rule.notifications_firefighter=1;}
else if (iRBR_in.health.fall_c==1 or
iRBR_in.health.fall_w==1)
{rule.notifications_caregiver=7;}
RBR_o.case =upro.ADL;
RBR_o.case_features =iCM_out;
RBR_o.rule =rule;
}
```

It should be noted that the CAMI architecture, the semantic encoding of its components are restricted to the scope of the verification, and hence the components like the Database, UI, Security and Privacy are not encoded as STA. The semantic encoding produces a complex NSTA comprising 32 STA, out of which 18 STA are produced by encoding the 10 AC of CAMI (4 sensors: one for detecting pulse data deviation, two for fall detection and one for fire detection, data collector, MQ, RBR, CBR, daily activity detection, fuzzy logic) and the remaining 12 by encoding 6 CC (Local Processor, Cloud Processor, DSS (Local and Cloud), Context modeling in DSS(Local and Cloud) of the AADL model of CAMI. On the other hand, the NSTA model of the minimum architecture configuration comprises of only 18 STAs and is shown to be scalable with exhaustive analysis.

10.7 AAL Architecture Verification and Discussion

In this section, we verify if the minimum configuration architecture, and the most complex one, the CAMI architecture introduced in Section 10.4, satisfy their requirements as described in the same section, respectively. We apply exhaustive model checking for the first case and statistical model checking in the second case.

Exhaustive verification of the minimum configuration using UPPAAL. The results of the exhaustive verification of the minimum configuration architecture using UPPAAL model checker are tabulated in Table 10.1. To check that our system meets its requirements, we employ a monitor STA that monitors the sensor values, the respective DSS output, and the corresponding clock. The monitor automaton for $R1_{\text{Arch1}}$ is shown in Fig. 10.11. As described, we start the monitoring clock *s1* when the pulse sensor produces the data, marked by transition to *L2* triggered by the synchronization channel and we stop the clock when a decision is produced by the cloud DSS. Similar monitors have been employed for $R2_{\text{Arch1}}$. We have used queries of the form *A leads to B* for our analysis and therefore a pre-check of each corresponding “A", being reachable is first carried out. Moreover, since our model is an STA model where each component has associated failure probabilities and failure of a component does not yield the intended results during exhaustive verification, we verify the properties considering all

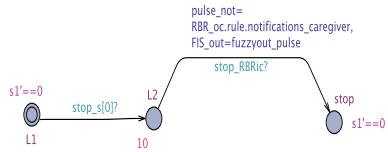


Figure 10.11: The monitor automaton for requirement $R1_{\text{Arch1}}$.

Table 10.1 UPPAAL analysis results for the minimum configuration architecture

Req.	Query	Result
R1_{Arch1}	$(110 \leq sd\_w.data\_val \leq 300\ and\ ADL = 1\ and$ $M\_pulse.FIS\_out == 3\ and\ op\_DC == 1$ $and\ op\_fuzzy == 1\ and\ op\_RBR == 1)$ $\rightarrow M\_pulse.pulse\_not == 3\ and M\_pulse.s1 \leq 20$	Pass
	$E <> (110 \leq sd\_w.data\_val \leq 300\ and\ and\ ADL = 1$ $M\_pulse.FIS\_out == 3\ and\ op\_DC == 1$ $and\ op\_fuzzy == 1\ and\ op\_RBR == 1)$	Pass
R2_{Arch1}	$(se\_w.fall == 1\ and\ op\_DC == 1$ $and\ op\_EU == 1\ and\ op\_RBR == 1)$ $\rightarrow M\_fall.fall\_not == 7\ and M\_fall.s1 \leq 20$	Pass
	$E <> (se\_w.fall == 1\ and\ op\_DC == 1$ $and\ op\_EU == 1\ and\ op\_RBR == 1)$	Pass

the components are operational. $R1_{\text{Arch1}}$ requires that if the pulse is high and the user is not exercising, then an abnormal pulse alert is raised to the caregiver within 20 s. In $R2_{\text{Arch1}}$, we verify that if the fall sensor detects a fall event, then a fall alert is raised to the caregiver within 20 s. The aforementioned requirements are safety requirements of the system and it is shown that these requirements are met provided all the system components are operational.

Statistical Verification of the CAMI architecture using UPPAAL SMC. In case of CAMI architecture, which is the most complex instantiation of our proposed generic architecture, exhaustive verification does not scale and hence we chose to verify the CAMI system requirements using UPPAAL SMC [5], the statistical extension of UPPAAL model checker to perform probabilistic analysis. To verify the functional requirements, we employ monitor STA to monitor the sensor values, the respective DSS output and the corresponding clock. For instance, an example of monitor STA for $R1_{\text{CAMI}}$ is given in Fig. 10.12. As shown, we start the monitoring clock $s1$ when the fire sensor

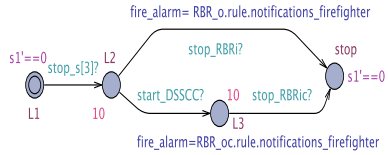


Figure 10.12: The monitor automaton for requirement $R1_{\text{CAMI}}$.

produces the data, marked by transition to $L2$ triggered by the synchronization channel and we stop the clock when a decision is produced by local DSS or the cloud DSS. Similar monitors are employed for $R2_{\text{CAMI}}$, $R3_{\text{CAMI}}$, $R4_{\text{CAMI}}$ and$R5_{\text{CAMI}}$.

The verification results are tabulated in Table 1. The CAMI architecture model satisfies all the requirements with probabilities close to 1 with a high confidence within 4 minutes until a result is returned. As in the other case, since most queries contain terms of the form $A\ imply\ B$, we first check the reachability of A. From the analysis, it follows that the probability of the cloud DSS to get activated (($R6_{\text{CAMI}}$) is [0.01, 0.04]. This is justified that it becomes active only when the local DSS has failed and the failure probability of local DSS is between [0.01, 0.04] for a simulation over 1000 time units, which is a safe value to assume for safety-critical systems.

Discussion. The approach presented in this paper paves the way for the development of formally assured future intelligent AAL solutions that integrate multiple functionalities. Our approach can be applied at earlier design stages to capture potential errors that can propagate across the development stages, which may result in significant re-engineering costs. Our architecture description framework (AADL) has a commercially available tool support, OSATE [28] for automated modeling, and provides some preliminary architecture-level analysis. It also allows us to model the behavior of the architecture components via behavior annex and encode the probabilities of failures of various components, via the error annex. However, AADL also has its limitations of expressing complex behaviors of algorithms such as CBR, which we have omitted in this work.

There are two analysis approaches presented in this paper: (1) using exhaustive model checking (2) using stochastic model checking, both automated automated via a commercial tools UPPAAL and UPPAAL SMC. The analysis approaches are chosen based on the system complexity. If the architecture model is scalable with exhaustive model-checjing, then it can be applied. Although the exhautive verification resulst are accurate, one cannot take into account the probabilistic behaviour of our systems. In case of complex models that needs to be analyzed for stochastic behaviours, the user can opt for simulation-based approaches, although it does not yield 100% accuracy. The verification results shown in this ppaper are specific to our architecture models defined, however one can use the approach to verify any set of requirements for various architecture types defined by the generic architectural model defined in this work. In

case of exhaustive model-checking, the resulst are derived assuming that all components are operational such that we devoid the system of its probabilistic failure behaviour. Also, for the case of statiscal model checking, it is worth mentioning that the results are derived assuming high reliability of individual architecture components and considering specific values for the periods and execution times. However, taking into account the wide variety of available sensors and other components, we can easily adapt the values to account for requirements of any specific architecture.

In addition, the approach presented in this paper is generic and easily extensible. Our modeling methodology based on AADL abstract components is easily extensible to suit particular run-time representations of the system. The AADL semantics as networks of STA is also generic and can be extended to accommodate other AADL properties that we have not accounted for in this work. We expect that similar results can be reproduced if the approach followed in this paper is used in other instances of integrated AAL solutions

Table 10.2 UPPAAL SMC Analysis Results for CAMI.

Req.	Query	Result	Runs
R1	$Pr[<= 1000]([](M\_fire.fire\_alarm == 1)\ imply\ (se\_nw.fire == 1$ and $M\_fire.s1 <= 20)))$	Pr [0.99975,1] confidence 0.998	3868
	$Pr[<= 1000](<>\ (M\_fire.fire\_alarm == 1))$	Pr [0.99975,1] confidence 0.998	4901
R2	$Pr[<= 1000]([](M\_fall.fall\_not == 7)\ imply\ ((se\_w.fall == 1$ or $sd\_nw.data\_val == 1)\ and (M\_fall.s1 <= 20))))$	Pr [0.99975,1] confidence 0.998	3868
	$Pr[<= 1000](<>\ (M\_fall.fall\_not == 7))$	Pr [0.99975,1] confidence 0.998	4901
R3	$Pr[<= 1000]([](M\_pulse.pulse\_not == 3)\ imply\ (110 <= sd\_w.data\_val <= 300\ and\ M\_pulse.FIS\_out == 3\ and\ ADL == 1\ and\ upro.disease\_history == 3$ and $M\_pulse.s1 <= 20))$	Pr [0.99975,1] confidence 0.998	3868
	$Pr[<= 1000](<>\ (M\_pulse.pulse\_not == 3))$	Pr [0.99975,1] confidence 0.998	3868
R4	$Pr[<= 1000]([](\ M\_firefall.fire\_not == 2\ and\ M\_firefall.fall\_not == 2\ imply\ ((se\_w.fall == 1$ or $sd\_nw.data\_val == 1)\ and\ se\_nw.fire == 1$ and $M\_firefall.s1 <= 20))$	Pr [0.99975,1] confidence 0.998	3868
	$Pr[<= 1000](<>\ (Pr[<= 100](<>\ (M\_firefall.fall\_not == 2$ and $M\_firefall.fire\_not == 2))$	Pr [0.99975,1] confidence 0.998	7905
R5	$Pr[<= 1000]([](M\_consistency.stop\ imply\ (RBR\_om == iCBRCC_m)))$	Pr [0.99975,1] confidence 0.998	3868
	$Pr[<= 1000](<>\ (M\_consistency.stop))$	Pr [0.99975,1] confidence 0.998	5777
R6	$Pr[<= 1000]([](INT\_CC.DSSCC\ imply\ PF\_DSS == 1))$	Pr [0.99975,1] confidence 0.998	3868
	$Pr[<= 1000](<>\ (INT\_CC.DSSCC))$	Pr [0.01,0.04] confidence 0.998	2885

10.8 Related Work

In recent years, there has been a lot of work in the area of AAL due to the need of supporting an increased elderly population [29]. Moreover, many functionalities that need to be tackled by AAL solutions are of a safety-critical nature, e.g., health emergencies like cardiac arrest, fall of the elderly, and home emergencies like fire at home, etc. [30], therefore work on their modeling and analysis is fully justified.

A study on existing AAL architectures shows that there are certain architecture types that address the construction of integrative AAL applications, some of the common ones being : Multi-Agent System (MAS) [31, 32, 33], Cloud-based [34, 35] and Internet-of-Things (IoT) centric [36].

- **Agent-based architectures:** These are the most commonly used architectures for AAL applications owing to its flexibility, autonomy, adaptability, better response and service continuity due to its distributed nature . Some examples of health care frameworks that relies on a distributed agent architecture are [37], [31]. However, the agent based architectures also have some drawbacks (i) Restricted communication protocols for agent communication and the delay overhead in taking a collective decision and (ii) maintaining the consistency of the framework .
- **Cloud-based AAL solutions:** AAL solutions that leverage the potential of cloud computing for context modeling, intelligent decision making and use it as a data storeAlthough cloud based solutions are scalable, cost-effective, reusable, adaptable, and extendable, the sole processing with cloud cannot guarantee strict hard real-time properties and the system fails completely in the absence of Internet.
- **IoT architectures:** IoT technology is now getting widely getting utilized in the filed of AAL owing to its technological advancements. The IoT concept of communication between smart objects and people and people are widely exploited in the field of AAL, thereby providing connectivity, context-awareness and adaptivity. . There are also approaches to integrate the autonomous behavior of agent-based systems with IoT technology [38, 39]. Although AAL systems based on IoT offer high flexibility, adaptability, the system depends only on the availability of the Internet for operation; which can lead to a complete failure of such systems in places where Internet connectivity is meager. Our architecture follows the design paradigms of Cloud-based AAL solutions, where the cloud is utilized for intelligent, context-aware decision making and as a data store, and is also augmented with local processing schemes to guarantee real-time properties. In many situations, cloud services cannot guarantee hard-real time properties and hence we adopted a local processing scheme as well in our model, and the cloud is a back-up which activates only when the primary has failed.

The formal assurance of AAL systems has been the focus of some related research in the recent years. Parente et al. provide a list of various formal methods that can be

used for AAL systems [40]. In another interesting work, Rodrigues et al. [4] perform a dependability analysis of AAL architectures using UML and PRISM. Other interesting research work uses temporal reasoning [3, 41] and Markov Decision Processes to formally verify the reliability of AAL systems [42]. Although these approaches target the formal analysis of AAL systems, most of the above work addresses only simple scenarios and are not used to analyze complex behaviors resulting from integrating critical AAL functions (e.g. fire and fall), as well as their decision making. In addition, these approaches do not aim to develop an overall framework for the verification of AAL systems, starting from an integrated architectural design, their design specifications, followed by a verification strategy, as proposed in this paper.

The use of Architecture Description Languages (ADL) to specify AAL designs has not been exercised previously, yet this is common when designing automotive or automation systems. There have also been approaches to formally verify AADL designs in other domains. The transformation approach from AADL to TA or variants has been already addressed by related work [43, 44, 45]. Although these approaches are automated verification techniques, there is a lack of focus on abstract components/patterns with stochastic properties. In addition, these approaches also suffer from state-space explosion, therefore they might not scale well to complex AAL designs. Nevertheless, there is interesting research that deals with stochastic properties and statistical model checking for the analysis of extended AADL models. One such example is in the work of Bruintjes et al. [46], where the authors have used SMC approach for timed reachability analysis of extended AADL designs. Although our approach also focuses on linear systems, it is different from the mentioned work in the fact that we focus on abstract components, and also introduce BA modeling for capturing the functional behavior of our modules, specifically for modeling the behavior of intelligent DSS. In their work, Bruintjes et al. use the SLIM Language, which is strongly based on AADL and is specific to avionics and automotive industry, including the error behavior and modes. However, we use the AADL core language with its standardized annex sets (EA and BA) for the architecture specification, thereby enabling us to represent the functional and error behaviour with the architecture model. The abstract component based modeling also brings exensiblity and reusability to our approach. Moreover, the authors only consider the event occurrences or delay variations using uniform or exponential distributions, whereas by employing our user-defined properties, we can also specify other distributions. Furthermore, the approach of Bruintjes et al. only deals with evaluation of time-bounded queries, however we also evaluate properties like reliability, data consistency, etc., along with timeliness. Another interesting work [6], possibly carried out in parallel with our work, employs statistical model checking using UPPAAL SMC to evaluate the performance of nonlinear hybrid models with uncertainty modeled in extended AADL. Although the approach is not specific to the AAL domain, it is promising to specify complex CPS systems considering uncertainties from physical environment. Unlike our model, the authors use Priced Timed Automata (PTA) models. In comparison, our approach considers only linear models that evolve continuously (yet

the analysis is carried out in discrete time due to sampling of continuous data). In brief, the two approaches resemble, yet our approach is all contained in the core language of AADL (as different from the mentioned work where the authors resort to other annexes integrated in OSATE), is tailored to systems that contain AI components, and assumes the random failure of various components, which is not considered in the related work.

10.9 Conclusions and Future Work

In this paper, we have proposed a generic AAL architecture and its intelligent Decision Support System that can tackle a multitude of functionalities by analyzing the interdependencies between simultaneously occurring events. We have also presented three specific instantiantions of the generic model, following an increasing order of complexity. In addition, we have also presented a framework for modeling and verification of our specific integrated AAL system architectures. To provide formal analysis for the AAL systems, we have semantically encoded the AADL model as NSTA model. These formal models has been shown to be analyzable exhaustively with UPPAAL or statistically with UPPAAL SMC, (chosen based on system complexity), to ensure that the required functional behavior is met. Our contribution is generic and paves the way for the development of formally assured intelligent AAL system architectures.

The framework is intended to augment existing AAL solutions with formal analysis support and provide analysis prior to implementation. Such an analysis is crucial in domains such as AAL, which are real-time, safety-critical, and require high levels of dependability. Due to the heterogeneity of components available in the AAL domain, the component failure probabilities, periods and execution times are not chosen w.r.t to any specific components, nevertheless the results presented in the paper are promising because the abstract components that have been proposed can be refined further.

In the future, we plan to enhance our DSS model with more rules for RBR and full functionality support of CBR and activity recognition, thereby providing an extensive analysis of AAL systems behaviors in possible critical scenarios. Another interesting direction to proceed with is providing automated tool support for the semantic mapping. We are also currently investigating on a distributed version of the integrated architectures for AAL, especially the one that supports multiple intelligent agents and its analysis.

Appendix A: AADL Model of RBR

```
1  abstract RBR
2  features
3  input: in event data port;
4  output: out event data port;
5  flows
```

```
6  F1 : flow path input -> output;
7  properties
8  Dispatch_Protocol => Aperiodic;
9  Compute_Execution_Time =>1s..1s;
10 end RBR;
11
12 abstract implementation RBR.impl
13 subcomponents
14 AAL_event: data System_Data_model::events;
15 DA: data System_Data_model: User_activity;
16 u_profile: data System_Data_model:User_profile;
17 fuzzy_out1:data System_Data_model::fuzzified_data_health;
18 fuzzy_out2: data system_Data_model::fuzzified_data_camera;
19 annex EMV2{**
20 use types error_model;
21 use behavior error_model::simple;
22 error propagations
23 input: in propagation{NoValue};
24 output: out propagation{ Novalue};
25 flows
26 ef0: error path input{NoValue}->output{NoValue};
27 component error behavior
28 events
29 Reset: recover event;
30 TF: error event;
31 PF: error event;
32 err_p: error event;
33 transitions
34 t0: Operational-[TF]->Failed_transient;
35 t1: Failed_transient-[Reset]->Waiting with 0.8,
36 Failed_Permanent with 0.2;
37 t2: Operational -[PF]->Failed_Permanent;
38 t3: Operational-[err_p]->Failed_p;
39 t4: Failed_p -[input]->Operational;
40 end component;
41 properties
42 EMV2::DurationDistribution => [Duration => 1ms..2ms;
43 Distribution =>Fixed;] applies to reset;
44 EMV2::OccurrenceDistribution =>[ProbabilityValue => 0.2;
45 Distribution => Fixed;] applies to Failure_Transient;
46 EMV2::OccurrenceDistribution =>[ProbabilityValue => 0.1;
47 Distribution => Fixed;] applies to Failure_Permanent;
48 **};
49 annex behavior_specification {**
50 states
51 Waiting: initial complete final state;
52 Operational: state;
53 transitions
54 Waiting -[on dispatch input]->Operational {if
55 (AAL_event="fire"){output:="notification_firefighter fire"}
56  elsif ( fuzzy_out1 = "Pulse_high" and DA!="exercising" and
57  u_profile="cardiac_patient")
58  {output := "notification_caregiver_highpulse"}
59 elsif (AAL_event = "fall" or fuzzy_out2 = "Fall_high")
60 {output := "notification_caregiver fall"}
61 elsif( fuzzy_out1 = "pulse-abnormal_low" )
62 {output:= "notification_caregiver"}
63 elsif(AAL_event = "fall" and fuzzy_out2= "Fall_high" and
```

```
64  AAL_event="fire" and fuzzy_out1= "pulse-abnormal_low")
65  {output:= "notification_caregiver fall,fire, pulse_low and
66  notification_firefighter fall, fire, pulse-abnormal-low"}
67  end if};
68  **};
69  end RBR.impl;
```

Acknowledgement

This work has been supported by the joint EU/Vinnova project grant CAMI, AAL-2014-1-087, which is gratefully acknowledged.

Bibliography

[1] Ashalatha Kunnappilly, Cristina Seceleanu, and Maria Lindén. Do We Need an Integrated Framework for Ambient Assisted Living? In *Ubiquitous Computing and Ambient Intelligence: 10th International Conference, UCAmI 2016, San Bartolomé de Tirajana, Gran Canaria, Spain, November 29–December 2, 2016, Part II 10*, pages 52–63. Springer, 2016.

[2] Ashalatha Kunnappilly, Alexandru Sorici, Imad Alex Awada, Irina Mocanu, Cristina Seceleanu, and Adina Madga Florea. A Novel Integrated Architecture for Ambient Assisted Living Systems. In *Computer Software and Applications Conference (COMPSAC), 2017 IEEE 41st Annual*, volume 1, pages 465–472. IEEE, 2017.

[3] Juan C Augusto and Chris D Nugent. The use of temporal reasoning and management of complex events in smart homes. In *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 778–782. IOS Press, 2004.

[4] Genaína Nunes Rodrigues, Vander Alves, Renato Silveira, and Luiz A Laranjeira. Dependability analysis in the ambient assisted living domain: An exploratory case study. *Journal of Systems and Software*, 85(1):112–131, 2012.

[5] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015.

[6] Yongxiang Bao, Mingsong Chen, Qi Zhu, Tongquan Wei, Frederic Mallet, and Tingliang Zhou. Quantitative performance evaluation of uncertainty-aware hybrid AADL designs using statistical model checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(12):1989–2002, 2017.

[7] Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical model checking: An overview. In *International conference on runtime verification*, pages 122–135. Springer, 2010.

[8] Peter H Feiler, Bruce Lewis, Steve Vestal, and Ed Colbert. An overview of the SAE architecture analysis & design language (AADL) standard: a basis for

model-based architecture-driven embedded systems engineering. In *Architecture Description Languages*, pages 3–15. Springer, 2005.

[9] RB Frana, J-P Bodeveix, Mamoun Filali, and J-F Rolland. The AADL behaviour annex–experiments and roadmap. In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 377–382. IEEE, 2007.

[10] Julien Delange and Peter Feiler. Architecture fault modeling with the AADL error-model annex. In *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, pages 361–368. IEEE, 2014.

[11] *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language*.

[12] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Information and computation*, 104(1):2–34, 1993.

[13] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium*, pages 414–425. IEEE, 1990.

[14] Kim G Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International journal on software tools for technology transfer*, 1(1-2):134–152, 1997.

[15] Peter E Bulychev, Alexandre David, Kim G Larsen, Axel Legay, Guangyuan Li, and Danny Bøgsted Poulsen. Rewrite-Based Statistical Model Checking of WMTL. *RV*, 7687:260–275, 2012.

[16] Feng Zhou, Jianxin Roger Jiao, Songlin Chen, and Daqing Zhang. A case-driven ambient intelligence system for elderly in-home assistance applications. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 41(2):179–189, 2011.

[17] Hamid Medjahed, Dan Istrate, Jerome Boudy, and Bernadette Dorizzi. Human activities of daily living recognition using fuzzy logic for elderly home monitoring. In *2009 IEEE International Conference on Fuzzy Systems*, pages 2001–2006. IEEE, 2009.

[18] UA651 BP sensor. `http://www.andmedical.com.au/products-service/value-ua-651`. Accessed: 2019-03-16.

[19] Fibaro motion sensor. `https://manuals.fibaro.com/content/manuals/en/FGMS-001/FGMS-001-EN-T-v2.0.pdf`. Accessed: 2019-03-16.

[20] Fitbit. `https://www.fitbit.com/se/home`. Accessed: 2019-03-16.

[21] Vibby fall detection sensors. `http://www.vitalbase.co.uk`. Accessed: 2019-03-16.

[22] CAMI Gateway. `https://eclexys.com/wp-content/uploads/2019/01/Exys9200-SNG-Brochure.pdf`. Accessed: 2019-03-16.

[23] Opentele. https://www.opentelehealth.com. Accessed: 2018-01-15.

[24] Linkwatch. https://www.linkwatch.se. Accessed: 2018-01-15.

[25] Tiago robotic platform. http://tiago.pal-robotics.com. Accessed: 2019-03-16.

[26] Pepper robot. https://www.softbankrobotics.com/emea/en/pepper. Accessed: 2019-03-16.

[27] Rabbit mq message broker. https://www.rabbitmq.com. Accessed: 2019-03-16.

[28] OSATE-Open Source AADL Test Environment. http://osate.github.io/. Accessed: 2018-05-15.

[29] Ruijiao Li, Bowen Lu, and Klaus D McDonald-Maier. Cognitive assisted living ambient system: A survey. *Digital Communications and Networks*, 1(4):229–252, 2015.

[30] Parisa Rashidi and Alex Mihailidis. A survey on ambient-assisted living tools for older adults. *IEEE journal of biomedical and health informatics*, 17(3):579–590, 2013.

[31] Juan De Paz, Sara Rodríguez, Javier Bajo, Juan Corchado, and Emilio Corchado. OVACARE: A multi-agent system for assistance and health care. *Knowledge-Based and Intelligent Information and Engineering Systems*, pages 318–327, 2010.

[32] David Isern, David Sánchez, and Antonio Moreno. Agents applied in health care: A review. *International journal of medical informatics*, 79(3):145–166, 2010.

[33] John Nealon and Antonio Moreno. Agent-based applications in health care. *Applications of software agent technology in the health care domain*, pages 3–18, 2003.

[34] Mobyen Uddin Ahmed, Mats Björkman, and Maria Lindén. A generic system-level framework for self-serve health monitoring system through internet of things (iot). *Studies in health technology and informatics*, 211:305–307, 2015.

[35] Abdur Forkan, Ibrahim Khalil, and Zahir Tari. CoCaMAAL: A cloud-oriented context-aware middleware in ambient assisted living. *Future Generation Computer Systems*, 35:114–127, 2014.

[36] Angelika Dohr, Robert Modre-Osprian, Mario Drobics, Dieter Hayn, and Günter Schreier. The Internet of Things for Ambient Assisted Living. *ITNG*, 10:804–809, 2010.

[37] Dante I Tapia, Sara Rodrıguez, and Juan M Corchado. A distributed ambient intelligence based multi-agent system for Alzheimer health care. In *Pervasive Computing*, pages 181–199. Springer, 2009.

[38] Giancarlo Fortino, Antonio Guerrieri, and Wilma Russo. Agent-oriented smart objects development. In *Computer Supported Cooperative Work in Design (CSCWD), 2012 IEEE 16th International Conference on*, pages 907–912. IEEE, 2012.

[39] Peter Leong and Liming Lu. Multiagent web for the Internet of Things. In *Information Science and Applications (ICISA), 2014 International Conference on*, pages 1–4. IEEE, 2014.

[40] Guido Parente, Christopher D Nugent, Xin Hong, Mark P Donnelly, Liming Chen, and Enrico Vicario. Formal modeling techniques for ambient assisted living. *Ageing International*, 36(2):192–216, 2011.

[41] using temporal logic and model checking in automated recognition of human activities for ambient-assisted living.

[42] Yan Liu, Lin Gui, and Yang Liu. MDP-based reliability analysis of an ambient assisted living system. In *International Symposium on Formal Methods*, pages 688–702. Springer, 2014.

[43] Loïc Besnard, Thierry Gautier, Paul Le Guernic, Clément Guy, Jean-Pierre Talpin, Brian Larson, and Etienne Borde. Formal semantics of behavior specifications in the architecture analysis and design language standard. In *Cyber-Physical System Design from an Architecture Analysis Viewpoint*, pages 53–79. Springer, 2017.

[44] Mohamed Elkamel Hamdane, Allaoui Chaoui, and Martin Strecker. From AADL to timed automaton-A verification approach. *International Journal of Software Engineering and Its Applications*, 7(4), 2013.

[45] Andreas Johnsen, Kristina Lundqvist, Paul Pettersson, and Omar Jaradat. Automated verification of AADL-specifications using UPPAAL. In *High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on*, pages 130–138. IEEE, 2012.

[46] Harold Bruintjes, Joost-Pieter Katoen, and David Lesens. A statistical approach for timed reachability in AADL models. In *Dependable Systems and Networks (DSN), 45th Annual IEEE/IFIP International Conference on*, pages 81–88. IEEE, 2015.

Paper D

Chapter 11

Paper D: Architecture Modelling and Formal Analysis of Intelligent Multi-Agent Systems

Ashalatha Kunnappilly, Simin Cai, Raluca Marinescu, Cristina Seceleanu.
In Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), May, 2019

Abstract

Modern cyber-physical systems usually assume a certain degree of autonomy. Such systems, like Ambient Assisted Living systems aimed at assisting elderly people in their daily life, often need to perform safety-critical functions, for instance, fall detection, health deviation monitoring, communication to caregivers, etc. In many cases, the system users have distributed locations, as well as different needs that need to be serviced intelligently and simultaneously. These features call for intelligent, adaptive, scalable and fault-tolerant system design solutions, which are well embodied by multi-agent architectures. Analyzing such complex architectures at design phase, to verify if an abstraction of the system satisfies all the critical requirements is beneficial. In this paper, we start from an agent-based architecture for ambient assisted living systems, inspired from the literature, which we model in the popular Architecture Analysis and Design Language. Since the latter lacks the ability to specify autonomous agent behaviours, which are often intelligent, non deterministic or probabilistic, we extend the architectural language with a sub-language called Agent Annex, which we formally encode as a Stochastic Transition System. This contribution allows us to specify behaviours of agents involved in agent-based architectures of cyber-physical systems, which we show how to exhaustively verify with the state-of-art model checker PRISM. As a final step, we apply our framework on a distributed ambient assisted living system, whose critical requirements we verify with PRISM.

11.1 Introduction

Equipped with various sensors, actuators and computation units, modern cyber-physical systems have evolved into increasingly intelligent, autonomous and adaptive systems. A representative category is Ambient Assisted Living (AAL) systems, which monitor the conditions of elderly people and their surroundings, in order to provide them with intelligent and timely assistance, autonomously. Due to such characteristics, as well as the possibly distributed locations of users and service providers, the multi-agent architecture is deemed appropriate for designing multi-user AAL systems. In a multi-agent system (MAS), each agent is an autonomous entity that can perform actions individually and intelligently, while adapting to the environment. For instance, a pulse agent may monitor an elderly user's pulse, and decide whether an alert should be sent to the caregiver. Multiple agents can be distributed geographically, and cooperate by exchanging network messages to achieve complex tasks, such as a proper reaction to the fall caused due to a sudden drop of pulse, via the cooperation of a pulse agent and a fall-detection agent. In many cases, such system behaviours are often probabilistic due to random component failures, communication failures, arbitrary service connection requests, user interactions, etc.

In order to guarantee the system's safety and achieve the desired quality of service (QoS), it is beneficial to ensure the correctness of the AAL system design, with respect to the real-time, fault-tolerant and probabilistic behaviors of agents, both individually and in cooperation. To achieve this, specification and rigorous analysis of such behaviors are necessary, which should go hand-in-hand with the specification and analysis of the entire architecture in which the agents are integrated. Existing techniques either do not support the integrated specification and analysis of architecture and agent behaviors, or lack reasoning capabilities of combined real-time, fault-tolerant and probabilistic behaviors that are essential to many AAL systems [1, 2].

In this paper, based on existing solutions [3], we propose a MAS architecture for AAL, comprising simple reflex agents based on if-then-else rules, and complex intelligent agents with self-learning based on Reinforcement Learning (RL) [4]. As our basis for specification, we choose a commonly-used architecture specification language, that is, the Architecture Analysis and Design Language (AADL) [5]. We use the original AADL constructs to specify the architecture including the agent components, their interfaces and communication. However, since the core AADL language lacks the ability to specify emergent agent behaviours, which may be both probabilistic and non deterministic, we propose an annex extension to the core AADL, referred to as **Agent Annex**. Unlike the existing Behaviour Annex specification of AADL [6], usually used for encoding component behavior, the Agent Annex allows one to describe the combined real-time, fault-tolerant and probabilistic behaviors. We formulate the new annex by extending the AADL meta model, and define its semantics as a stochastic transition system. To enable formal verification, we also provide formal semantics to the AADL architectural model, in terms of Stochastic Transition Systems (STS). We employ the

state-of-the-art probabilistic model checker, PRISM [7], to formally verify a set of crucial functional and quality-of-service properties of an illustrative AAL use case.

The rest of the paper is organized as follows. Section 11.2 overviews the basics of AADL, STS and PRISM. In Section 11.3, we describe our AAL system architecture based on MAS. We present the AADL modeling constructs and the Agent Annex extension in Section 11.4. Section 11.5 describes the formal encoding of the AADL model, and in Section 11.6, we present the verification results applying the PRISM model-checker on a representative AAL system. Related work is described in Section 11.7. Some discussion points are presented in Section 11.8 and conclusions and future work in Section 11.9.

11.2 Preliminaries

In this section, we give an overview of AADL, STS and PRISM, in Sections 11.2.1, 11.2.2 and 11.2.3, respectively.

11.2.1 Architecture Analysis and Design Language

The Architecture Analysis and Design Language (AADL) [5] is a textual and graphical language for modeling and analyzing a real-time system's hardware and software architecture as hierarchies of components at various abstraction levels.

AADL component categories like *Application Software*, *Execution Platform* and *System* are used to represent the run-time architecture of the system, whereas a more generalized representation is also possible by specifying it as *abstract*. A component in AADL can be defined by its *type* and *implementation*; the first defines the interface of the component and its externally-observable attributes, whereas the second defines its internal structure. AADL allows possible component interactions via *ports/features*, *shared data*, *subprograms*, and *parameter connections*, whereas a communication protocol over a network connection is modeled by a *bus*. The components can also be associated with various *properties*, like *period*, *execution time*, and *dispatch protocol*. The dispatch protocol specifies if the component trigger is *periodic* or *aperiodic*. We also employ various user-defined properties for representing the probabilistic distribution of an aperiodic event and the rate at which a component recovers from the failure. All the AADL declarations are declared in packages and are therefore accessible to other packages, or they can be declared directly in an AADL specification and not be accessible to packages.

The AADL core language is designed to be extensible and can be extended via user-defined properties and annex sub-languages. User-defined properties are relatively simpler extensions, when compared to sub-languages, and can be associated with modeling elements as simple values, for instance, integers or strings. However, sub-languages allow more complex structures to be added to an AADL model. A sub-language can

be standardized and published as an AADL annex. Several such annexes have been defined, for example, the *behavior annex* to model the component's behaviour, and the *error annex* for modeling the error behaviour of the system. Annex sub-languages are included into AADL specifications as annex libraries or annex subclauses. An annex library is used to define classifiers defined in an anonymous namespace, or in a public or private part of a package. Annex subclauses are inserted into component types and component implementations and can reference the classifiers declared in the annex library. In AADL, annexes are considered to be separate from the core AADL, i.e., if we remove all the annex libraries, subclauses, and annex-related property associations, the resulting model is a valid core AADL model. For further details, the reader can refer to the work [5].

11.2.2 Stochastic Transition Systems

Stochastic transition systems (STS) [8] are transition systems that support non determinism, and transitions with unspecified delay distributions, providing concise and compositional means to represent systems in terms of probability, waiting-time distributions, non determinism, and fairness.

A *stochastic transition system* is defined by a tuple $S =< V,\ \Theta,\ T >$, where $V = V_{\mathrm{l}} \cup V_{\mathrm{g}}$, V_{l} is a finite set of $local\ state\ variables$ with finite domain, and V_{g} is the finite set of *global variables* of the system. In case a subset of V_{g} is used in a particular module, i.e., $V_{\mathrm{g}} \cap V_{\mathrm{l}} \neq \emptyset$, implies that V_{g} also contributes to the state-space of the module. We denote by $s[\![v]\!]$ the value in state $s \in S$ of $v \in V_{\mathrm{l}}$ (the interpretation of function $[\![\cdot]\!]$ is extended to terms in the obvious way). Θ is an assertion over V_{l} denoting the set $\{s \in S \mid s \models \Theta\}$ of $initial\ states$, and the assertions over V_{g}. T is a set of $transitions$. The following quantities are associated with each transition $\tau \in T$:

- An assertion ϵ_τ over V_{l}, which specifies the set of states $\{s \in S \mid s \models \epsilon_\tau\}$ on which τ is enabled.
- A number m_τ of transition modes, where each transition mode $i \in \{1, \ldots, m_\tau\}$ corresponds to a possible outcome of τ. Each transition mode i is specified by V_{l}: (i) a set of assignments $\{v' := f^\tau_{i,v}\}_{v \in V_{\mathrm{l}}}$, where each $f^\tau_{i,v}$ is a term over V_{l} and $f^\tau_i : S \mapsto S$ is a function that maps every state $s \in S$ to a successor $s' = f^\tau_i(s)$ such that $s'[\![v]\!] = s[\![f^\tau_{i,v}]\!]$ for all $v \in V_{\mathrm{l}}$, and (ii) the probability $p^\tau_i \in [0, 1]$ with which mode i is chosen, where $\sum_{i=1}^{m_\tau} p^\tau_i = 1$.

The set of transitions T is partitioned into the set of $immediate\ transitions\ T_i$ and the set of $delayed\ transitions\ T_d$. Immediate transitions must be taken as soon as they are enabled, and a subset of these transitions $T_f \subseteq T_i$ is the set of $fair\ transitions$. In turn, the set of delayed transitions is partitioned into: (i) the set of transitions with $exponential\ delay\ distribution\ T_e$, where for each $\tau \in T_e$ there is an associated transition rate $\gamma_\tau > 0$, and (ii) the set of transitions with $unspecified\ delay\ distributions\ T_u$ that are taken with non-zero delay, but the prob-

ability distribution of the delay and the possible dependencies between this distribution and the system's state or past history are not specified.

Given a state $s \in S$, we indicate by $T(s) = \{\tau \in T \mid s \models \epsilon_\tau\}$ the set of transitions enabled by s. To ensure that $T(s) \neq \emptyset$ for all $s \in S$, an $idle\ transition\ \tau_{idle}$ is added to every STS defined by $\epsilon_{\tau_{idle}} = true$, $m_{\tau_{idle}} = 1$, $p_1^{\tau_{idle}} = 1$, $\gamma_{\tau_{idle}} = 1$ and by the set of assignments $\{v' := v\}_{v \in V}$.

11.2.3 Probabilistic Timed Automata and PRISM

To analyze our multi-agent systems, in this paper we use the PRISM model checker [7]. Among other supported formal notations, PRISM provides symbolic model checking of systems modeled as networks of Probabilistic Timed Automata (PTA), which are semantically described by Timed Probabilistic Systems (TPS)[9]. De Alfaro shows that an STS can be straightforwardly translated into (fair) TPS, yielding the same state space [8].

In PRISM, a PTA is represented by a *module*, which is defined as a tuple $M =< Var, Clock, C >$, in which Var is a set of local finite-valued variables, $Clock$ is a set of local clock variables that progress with step of 1, and C is a set of *commands*. The state of a PTA is the valuation of $Var \cup Clock$. The commands, which define the transitions of the system, are specified as guarded probabilistic updates of states in the following form: $[a]\, g \rightarrow p_1 : u_1 + ... + p_n : u_n$. Here, guard g is a predicate over the variables that enable the transition. Variables $p_1, ..., p_n$ are probabilities within the interval (0,1], whose values sum up to 1. Each u_i is an update of the state by assigning new values to variables, or by resetting clocks. The update of a variable v is specified as $v' = n$, where n is the new value. A command is enabled if the guard of the command evaluates to true. If multiple commands are enabled, one command is selected non-deterministically, and one of its updates is executed probabilistically. In the brackets, a is a labeled action. Commands with same actions are forced to be taken simultaneously. We can also augment the model with *rewards*, which are real values associated with states or transitions. Rewards can be both positive or negative depending on the system behaviour.

A *system* is defined as a network of modules via parallel composition: $Sys = M_1||...||M_n$. A global state is the valuation of all variables of all modules. A module can both read and write its own local variables, but only has read access to the local variables of other modules. Synchronized transitions of modules are identified by the commands with the same labels.

The property specification language of PRISM for PTA is based on Probabilistic Computation Tree Logic (PCTL) [10]. The model checker can verify whether the probability of a path property pp is within a bound b, which is specified as: $P\, b\, [\, pp\,]$. Here, b can be any of $>= p$, $> p$, $<= p$ or $< p$, where p is a double within [0,1]. A path property pp is a formula that evaluates to either true or false for a single path in the model, in which one can apply the following operators: X (next), U (until), F (eventu-

ally), G (always), W (weak until), R (release). PRISM can also compute the minimum and maximum probabilities of a path property, in the form of: $Pmin =?[\,pp\,]$, and $Pmax =?[\,pp\,]$, respectively. In order to check a path property for paths that start from multiple states, *filters* are used to identify the starting states. For instance, the "forall" filter returns true if the property is true for all states satisfying the filter.

In the following section, we present a multi-agent system (MAS) architecture for the AAL domain.

11.3 A Multi-Agent System Architecture for AAL

Our proposed architecture consists of multiple agents, and ensures improved fault-tolerance, scalability and adaptability, compared to centralized architectures in the domain, such as CAMI [11]. The architecture is inspired from similar existing architectures in literature [3]. However, existing solutions usually suffer from additional overhead encountered during agent synchronization for collective decision-making and data consistency maintenance. This overhead can sometimes hamper the real-time behavior of the system. Hence, we investigate how we can use these systems for developing integrated solutions that ensure a safe trade off between autonomous behavior and consistency overheads. This is challenging since agents are interdependent, and have only a limited view of the environment. Concretely, the agent-based solution should ensure a consistent view of the environment, in terms of processed data and events, as well as an inter-agent communication overhead that should not result in breaching the real-time system demands. We ensure this by allowing each agent to cater for a particular functionality, respectively; for instance, a health-monitoring agent detects health-parameter variations and raises a notification to caregiver. However, in order for the agents to cooperate in real time, each agent maintains the dependencies it can have with other agents, in a list that can change at run time [1].

The architecture is described briefly in the following, and is shown in Fig. 11.1. It consists of the following components:

- **Agents**: In our solution, each agent tackles a particular functionality, in response to the sensor data, that is, the *fire agent* deals with detecting fire events from fire sensors and sends a notification to firefighters, the *pulse agent* detects the pulse data variations and sends a notification to the caregiver, the *fall agent* detects the user fall and alerts the caregiver, the *exercise agent* schedules and monitors the exercise session of the user, etc. These agents can belong to different categories, ranging from simple *reflex* agents to complex *intelligent* agents. In our case, we use the exercise agent as an example of an intelligent agent with embedded reinforcement learning (RL) algorithms. This provides an optimized exercise

[1] This claim is based on the simulation of the AADL model of the architecture for end-to-end latency according to the process detailed here: https://github.com/ashalatha-0504/Real-time-behvaiour-of-MAS

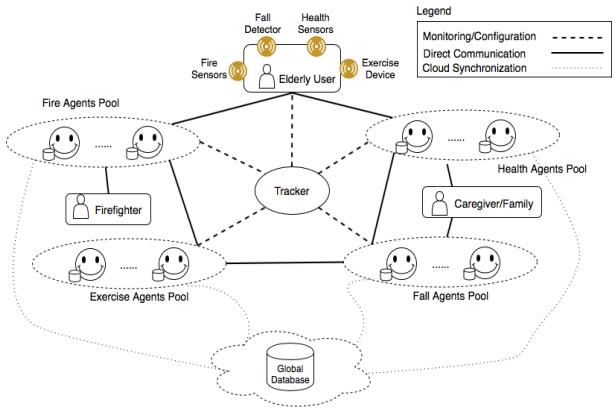


Figure 11.1: A MAS Architecture for AAL

session for the user, taking into account his/her health condition, preferences and exercise trends. All other agents are modelled as reflex agents encoded using "if-then-else" rules to handle the particular scenario. To be able to cooperate efficiently in real time (by reducing extra overheads), each agent is equipped with a list of possible dependencies with other agents. For instance, a fall agent has a dependency relation with a pulse agent. If a heavy fall occurs, the fall is first communicated to the caregiver, and then the fall agent synchronizes with the pulse agent to see if there are any pulse deviations (like a low pulse). If a low pulse is detected, the fall agent updates its notification to the caregiver indicating that the fall may be due to a low pulse. Each agent also maintains a small local database to store the individual data and keep track of the processed events and the decisions taken. The dependency lists are also maintained in the local database.

- **Tracker**: The system has a tracker that keeps the record of the IP addresses of all the agents in the system. The user's connections to the agents are established via the tracker. If the tracker fails at any point in time, the system continues to function via direct connections between user requests and agents. As shown in Fig 11.1, we have multiple agents of each category, which can accept requests from multiple users, arbitrarily, based on availability.
- **Cloud Database**: We also maintain a large-scale database in the cloud. All the local databases of the agents eventually synchronize with this cloud database. The cloud database also maintains the domain information about the user, like age, disease history, user preferences, etc.

- **End users of the system**: There are two types of users, elderly users and the service providers (caregivers, firefighters, etc.).

  We assume the following: (i) Each of the agents can accept a maximum of m connections, and there is a maximum of n users of the system, (ii) The number of accepted connections is always smaller than or equal to the number of users, that is, $m \leq n$, (iii) The system components communicate via various network protocols, (iv) The communication between agents is mediated by the tracker and is assumed instantaneous; however, if the tracker fails, then the agents can communicate to each other with an assumed delay.

11.3.1 Reinforcement Learning in Exercise Agents

The interaction between the exercise agent and its environment is modeled as an RL problem as follows: An exercise agent proposes 2 kind of exercise categories for its user - Low-intensity, and Medium-intensity, specifically tailored for cardiac patients, and normal users, respectively. Each category has a set of individual exercises. If a calendar notification is raised for the start of the exercise session, the exercise agent becomes operational and communicates with the health agent to see if the user has a normal pulse range. If the pulse level is normal, the health agent is ready to propose an optimized exercise session for the user. At any point in time, the exercise agent has 2 choices to make: a) choose an exercise category out of the 2 options, and b) suggest an exercise duration. The choice is made initially by considering user preferences and health condition. For simplicity, we assume that both options for exercise sessions are initially set to 10 minutes. The exercise duration is subdivided into intervals of 5 minutes. In these sub-intervals, the user gets an exercise recommendation of the same category. If the user quits in between (or not satisfied), the exercise category is re-adjusted in the next sub-interval. For each of the action that the agent suggests, it gets a reward back, based on the utility of the suggested action. The utility is calculated as a weighted sum of the following parameters: (i) user satisfaction for the prescribed exercise (*u_st*), based on a user feedback recorded and (ii) session completion, that is, if the user has completed the prescribed exercise duration (*ss_com*):

$$Utility = w1 * (u\_st) + w2 * (ss\_com), \tag{11.1}$$

where w1, w2 are the respective weights, where w1>w2. In this case, we assume that these weights are assigned to 0.6 and 0.4.

$$u\_st = \begin{cases} 1 & \text{if user satisfied} \\ -1 & \text{if user not satisfied} \end{cases} \tag{11.2}$$

$$ss\_com = \begin{cases} 1 & \text{if exercise duration completed} \\ -1 & \text{if exercise duration not completed} \end{cases} \tag{11.3}$$

In this paper, we consider that the **reward** signal is directly proportional to system utility, i.e, we get a higher reward for taking an action with higher utility. For this purpose, the initial system reward is calculated as its utility. After this, we always add up the successive reward values to determine the cumulative reward. In our case, we calculate the cumulative reward for each of the chosen exercise category, and the best action is considered as the one that has the maximum cumulative reward at any time point. In addition to the reward function, we also take into account the domain knowledge to make the choice of the exercise. The domain knowledge in our case consists of the user disease history, and preferences. It should be noted that the initial choice of exercise is made based on domain knowledge and thereafter, the choice is made by comparison of the reward variables., i.e., an exercise of a higher reward is always weighted over the other choice.

11.3.2 Use-Case Scenarios and System Requirements

In this paper, we consider a MAS consisting of a pulse agent, a fire agent, a fall agent, and an exercise agent, each with its replica, respectively. Each agent can accept a maximum of 2 connections, while the system is simultaneously utilized by two elderly adults, say Jim and Mary, living independently in their respective homes. Jim is also a cardiac patient. We consider the following two scenarios where the AAL system assists its users.

- *Scenario 1: Fall due to a low pulse:* The pulse-detection sensor worn by Jim detects the low pulse, and the wearable fall-sensor detects the fall. The sensors forward the sensed data to the tracker, which assigns a pulse agent and a fall agent to user 1 (arbitrarily, based on availability). The agents communicate with each other and reach the conclusion that the fall is due to a low pulse, and send a notification alert to the caregiver.
- *Scenario 2: Fire and Fall occurring simultaneously:* Mary is cooking dinner, and suddenly she feels dizzy and falls. The cooker is still on, starting a fire at home. In this case, the sensors alert the tracker of the respective events, and the former assigns a fire agent and a fall agent to Mary. The agents communicate with each other, synchronize the simultaneous occurrences of both events, and alert both the firefighter and the caregiver.
- *Scenario 3: Health abnormality during the exercise session:* Jim gets a calendar notification to start the exercise session. The tracker then assigns an exercise agent to Jim to schedule and monitor the exercise session. The exercise agent communicates with the health agent and identifies that Jim's health is normal and suggests the medium-complexity exercise for cardiac patients based on his preferences and health condition. In the middle of the exercise session, Jim's health agent indicates a sudden increase in pulse and hence the exercise agent suggests an exercise of lower complexity in the next sub-interval. The following system requirements are formulated for the above scenarios:

- R1: If a fall occurs due to low pulse, then raise an alert to caregiver indicating *fall due to low pulse* within 20 s. It relates to Scenario 1.
- R2: If a fire and a fall event occur simultaneously, then raise an alert to both caregiver and firefighter indicating the issue, within 20 s. This requirement relates to Scenario 2.
- R3: The exercise session is scheduled only if the health agent indicates a normal pulse.
- R4: The initially suggested exercise is based on user preferences and health condition.
- R5: If any health abnormality is detected in the first sub-session of the exercise, a different set of exercises of lower intensity is prescribed. Requirements R3, R4 and R5 are formulated based on Scenario 3. It should be noted that R1-R5 are safety-critical requirements.

In addition, the system has quality-of-service (QoS) requirements as follows:

- R6: If the tracker fails, the system continues its functionality.
- R7: If one of the agent fails, its function is carried out by the back-up.

11.4 Modeling Multi-Agent Systems in AADL

In this section, we illustrate the AADL modeling of our MAS, depicted in Fig 11.1. The components are modeled as follows: the agents and tracker are modeled as abstract components, which can be extended to suit a hardware or software implementation, at later stages of design. The sensors are modeled as hardware devices. The databases are modeled as data components in AADL. All the components have their respective component type and implementation defined. The component type defines the component features and properties. We use bus connections to represent the respective communication protocols used by the components. The bus access is modeled as a feature of the component. We restrict to only using properties like dispatch protocol, period, execution time and user-defined properties to specify the scope of global variables in the system. However, based on the requirements, certain user-defined properties can also be added to specify the rate of occurrence of an aperiodic event or so [1]. In the component implementation, we define the sub-components and connections.
Listing 11.1 shows an excerpt of the AADL model of our system with an exercise agent, and a bus component; the Agent Communication Protocol (ACP) models the communication protocol between multiple agents. For simplicity, we assume that the communication protocols defined here work via shared variables. The *Agent* component is modeled as an abstract component in AADL (Lines 1-8), which can be later refined towards a particular hardware or software, based on the application. We also show a

system-level representation (Lines 10-26) with its sub-components, user-defined properties (some of which needs assertion in the respective agent annex, where the property is applied) and their connections defining the communication.

Listing 11.1: An excerpt of the system modeling in AADL

```
1  abstract Exc_Agent1
2    features
3      BA1: requires bus access ACP;
4      BA2: requires bus access SA_comm1;
5      properties
6          Dispatch_Protocol => Aperiodic;
7          Compute_Execution_time => 2ms..2ms;
8  end Exc_Agent1;
9  bus ACP ... end ACP;
10 system agent_system
11 properties
12  mypropetries:: w1=0.6 applies to Exc_Agent1;
13  mypropetries:: w2=0.4 applies to Exc_Agent1;
14  myproperties:: utility  => "The value needs assertion in annex!"
15  applies to Exc_Agent1 ;
16   myproperties:: reward =>
17   "The value needs assertion in annex!"
18   applies to Exc_Agent1;
19 end agent_system;
20 system implementation agent_system.impl
21   subcomponents
22   A1: abstract Exc_Agent1;
23   Agent_Comm_Proto: bus ACP;
24   connections
25     BAsys1: bus access Agent_Comm_Proto <->A1.BA1;
26 end agent_system.impl;
```

After specifying the components and their interfaces, the next step is to specify the behaviour of the agent system. In the following sub-section, we propose an AADL annex specification tailored to modeling the autonomous behaviours of multi-agent systems and their learning algorithms.

11.4.1 Modeling Behaviours of Agents in AADL: Agent Annex

We present the syntax and semantics of our proposed *Agent Annex*, the AADL extension that we introduce in order to encode behaviors of agents.

Metamodel extension of AADL. The structure of our *Agent Annex* is defined by extending the AADL metamodel [12], represented as UML2 class diagrams. All classes in the Agent Annex metamodel are defined as subclasses of class *AObject*, the root class of the AADL metamodel. *Named objects* in the Agent Annex model is a subclass of the *Property Holder* class, allowing an object to have a name and associated AADL properties. Abstract classes in the metamodel are tagged by an *"A"*. The *Agent Model Annex* is formulated by extending the AADL abstract classes, *Annex Library* and *Annex subclause*. All the expressions of the Agent Annex are introduced as subclasses of these

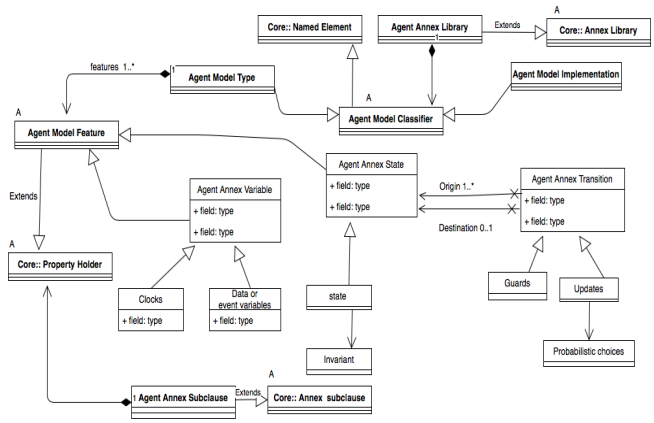


Figure 11.2: Agent Annex Metamodel

abstract classes. The *Annex Library* is used to declare *classifiers* of our Agent Annex in packages. The Annex Library concepts are attached to an AADL model by using the *Annex subclause* within a component type or component implementation declaration. An *Annex subclause* can refer to items in the *Annex Library*, and to basic AADL model elements. The Agent Annex metamodel is presented in Fig. 11.2.

Like all other components, Agent Annex also has a *type* and an *implementation* classifier. The Agent Model feature includes the definitions of *annex variables*, *states* and *transitions*. Any specific kind of variable, including clocks, can be declared in the Agent Annex. The state of an Agent Annex can also be associated with an *invariant*. The transitions are defined by using guards and updates. The updates support probabilistic choices.

Semantics and Syntax of Agent Annex. The Agent Annex (AA) is formally encoded as an STS, as follows:

$$AA = \langle Var, Init, Tt \rangle \tag{11.4}$$

- where: Var represents the set of local and global variables defined in the AA ;
- $Init$ is the assertion over Var denoting the set of initial states, formulas and also specifies association of reward values to a state or transition ;
- Tt is the set of state transitions, defined accordingly as in Section 11.2.3.

An excerpt of the Agent Annex subclause of the exercise agent is defined in Listing 11.2. As shown in Listing 11.2, the annex defines a probabilistic transition system with 7 states - $Idle$, Op, $Comm$, Exc_sc, $Ex1$, $Ex2$ and $Fail$, and a clock variable x.

$Idle$ represents the initial state. It also defines a probabilistic transition from state $Idle$. The transition is enabled aperiodically based on the calendar schedule for exercise and it has a probability of 0.999 to reach the state Op, and of 0.001 to reach the state $Fail$. Lines 11-21 define the other transitions specific to exercise agent. For instance, Lines 11-13 define the transitions for initiating communication with the user's pulse agent. If the agent reaches the $Comm$ state, it will initiate communication with the pulse agent and the exercise session starts only if the user pulse is normal. Lines 16-23 illustrate the exercise recommendation based on RL. Upon reaching the Exc_sc state initially, an exercise recommendation is made to the user based on the user preferences and disease history. The exercise duration is 10 min split in 2 intervals of 5 min each. Upon an initial recommendation (say category 1), the agent moves to the state $Exc1$ (Lines 14-16). The agent stays in this state until the completion of split duration of 5 min or until the user has decided to quit the exercise session. If the exercise split interval is less than 2 and greater than 0 (assuming the duration is 10 min), the exercise schedule has to continue and in the next split interval the agent recommends the action with the highest reward (Lines 20-21). The associated variables and their assertions are defined in the variables section Lines 22-30. Lines 28-30 indicates that the sys_rew1 is associated with the state $Exc1$ depending on user satisfaction or session completion and also with the transition $r1$ defined by lines 17-19. Similarly, there is sys_rew2 calculated for exercise 2, however due to space constraints, we do not show transitions for exercise category 2.

Listing 11.2: An example of Agent Model Annex Subclause attached to Exercise Agent

```
1  system implementation exc.agent
2    subcomponents
3    exercise_sensor: device exc_sensor;
4    annex Agent_Model {**
5       states
6       Idle, Op, Comm, Exc_sc, Exc1, Exc2, Fail;
7       Idle: initial state;
8        transitions
9        []state=Idle  & cal_exc=1->0.999:
10       (state'= Op & x'=0) + 0.001:(state'=Fail);
11       []state=Op & x= exe-> state'= Comm & x'=0;
12       []state= Comm & h_stat = 1 -> state'
13       = Exc_sch  & x'=0;
14       [] state = Exc_sch & u_pre=1 & d_his=0 &
15       exc_split=0 -> (state'=Exc1) & (exc_rec'=1) &
16       & (exc_split'=2) & (x'=0);
17       [r1]state= Exc1 & (x=5| u_quit=1) & h_stat =1
18       &  exc_split < 2 -> (s3'=5) & (exc_split'=
19       exc_split+1) & (x'=0);
20       []state=Exc_sch & exc_split <2 & exc_split >0
21       &sys_rew1> (sys_rew2) -> (state'=Exc1);
22         variables
23          bool cal_exc; bool u_quit; bool ss_com;
24          int exc_rec; clock x;
25         formula utility1 = w1*(u_st)+w2*(ss_com);
```

```
26          formula sys_rew1= utility1;
27          formula sys_rew1= sys_rew1+utility1;
28          reward_ass state=Exc1 & (u_sat=0| u_sat=1) &
29        (ss_com=1|ss_com=0) : sys_rew1;
30        reward_ass [r1] true : sys_rew1; **};
31  end Exc.agent;
```

In the following section, we define the syntax and semantic encoding of a complete AADL component, consisting of its interface and agent annex, and discuss its semantic mapping to an STS.

11.5 Formal Encoding of MAS

The first step of encoding our multi-agent architecture formally is to assign formal semantics to the specific AADL components that we utilize for modeling our system. An AADL component employed in this paper is defined by the following tuple:

$$AADL_{\text{Comp}} = \langle Comp_{\text{type}}, Comp_{\text{imp}}, AA \rangle, \tag{11.5}$$

where $Comp_{\text{type}}$ is the component type, $Comp_{\text{imp}}$ represents the component implementation, and AA, the agent annex specification [2].

- $Comp_{\text{type}}$ is in turn defined as a tuple: $Comp_{\text{type}} = \langle Features, Prop \rangle$, where:
  - $Features$ model the bus access that abstracts the communication protocol utilized by the system.
  - $Prop$ lists the associated properties of the component, like $Deployment$, $Communication$, $Timing$, $Thread\text{-}related\ properties$, etc. In this work, we only consider a subset of $Timing$ and $Thread\text{-}related\ properties$, as follows: $Prop = \{T_{\text{p}}, T_{\text{e}}, Dispatch\ protocol\}$, where T_{p} and T_{e} represent the period and execution-time of the component, respectively, T_{p}, $T_{\text{e}} \in Timing\ properties$, $Dispatch\ protocol \in \{P, AP\}$, where *P* and *AP* represents periodic activation and aperiodic activation, respectively.
- $Comp_{\text{imp}}$ is defined as $Comp_{\text{imp}} = \langle SC, Con \rangle$, where:
  - SC represents the sub-components of the system,
  - Con represents the set of connections. The function $F_{\text{con}} : Con \rightarrow Features$ assigns $Features$ to Con.
- Agent Annex AA follows the semantics defined in Section 11.4.1.

[2] Although Agent Annex is specifically tailored to represent agent behaviours, it can also specify the behaviours of other components, like the standard Behaviour Annex.

Table 11.1 Encoding of AADL as STS.

AADL	STS
$\langle Comp_{\text{type}}, Comp_{\text{imp}}, AA\rangle$	STS
T_{p}	$Invariant + Gaurd$
T_{e}	$Invariant + Guard$
Bus	$Variable$
$Data$	$Variable$
$Sub-components$	STS
$AA\ states$	$Variables$
$AA\ transitions$	$Transitions$
$AA\ variables$	$Variables$
$System$	$\|\|_{i=0}^{n} STS$

Definition 3. *The AADL component defined by Equation (2) is formally encoded as an STS. The MAS architecture is represented as a parallel composition of all the STS modules:* $MAS = ||_{i=0}^{n} STS_{modules i}$*, where* n *is the number of AADL components of the system, excluding data components and bus components, if defined in the system (as variables in the AADL component using them). The STS modules elements are as follows:*

- V *is the set of states of AA, defined by the values of all variables in AA (assuming that all the required variables including data/events, communication, and output variables have a local copy maintained in the corresponding AA of the component), and the reward variables if needed to specify the RL behaviour. The clock variables values are given by the component's period and execution-time properties of* $Comp_{type}$ *definition.*
- Θ *denotes the initial states encoded as an assertion over* V*.*
- T *represents the set of transitions defined in AA.*

The formal encoding is tabulated in Table 11.1. We now present an example of the above formal encoding, by applying it on a fire agent of our use case. The fire agent is formally encoded as an STS module, where:

- $V : \{(s1 = 0, fire = 0, fire\_alarm = 0, x = 0), (s1 = 0, fire = 0, fire\_alarm = 0, x = 1), (s1 = 1, fire = 1, fire\_alarm = 1, x = 0), (s1 = 1, fire = 1, fire\_alarm = 1, x = 1), (s1 = 1, fire = 1, fire\_alarm = 1, x = 2), (s1 = 2, fire = 1, fire\_alarm = 0, x = 0)\}$
- $\theta : s \models (s1 = 0 \wedge fire = 0 \wedge fire\_alarm = 0 \wedge x = 0)$

- T is defined by the set of transitions as follows:
  $T : \{\tau 1 : \{(s1 = 0 \wedge fire = 0 \wedge fire\_alarm = 0 \wedge x = period) \longrightarrow (s1' = 0 \wedge fire' = 0 \wedge fire\_alarm' = 0 \wedge x' = 0), P = 1\},$
  $\tau 2 : \{(s1 = 0 \wedge fire = 1 \wedge fire\_alarm = 0 \wedge x = period) \longrightarrow (s1' = 1 \wedge fire' = 1 \wedge fire\_alarm' = 1 \wedge x' = 0), P = 0.999 \cup (s1' = 2 \wedge fire' = 1 \wedge fire\_alarm' = 0 \wedge x' = 0), P = 0.001\},$
  $\tau 3 : \{(s1 = 1 \wedge fire = 1 \wedge fire\_alarm = 1 \wedge x = 2) \longrightarrow (s1' = 0 \wedge fire' = 0 \wedge fire\_alarm' = 0 \wedge x' = 0), P = 1\}\}$

Similarly, all other AADL components are encoded as STS modules, respectively. In the next section, we describe our verification approach with PRISM.

11.6 System Analysis with PRISM

The STS modules are encoded as a set of PTA modules in PRISM. The architecture is a parallel composition of the PTA modules. Each agent can accept at most 2 connections, and each has a redundant copy. Therefore, in order to ensure parallel processing, we assume 4 PTA for a single category of agent. Thus, we have 16 agent PTA that deal with pulse monitoring, fall monitoring, exercise monitoring and fire monitoring. In addition, we have one tracker PTA, through which connections between the agents are established. The sensor data and internal databases are modeled as variables, and their communication is modeled via shared data access. For simplicity, we have not chosen to model the cloud database.

Listing 11.3 shows an excerpt of exercise agent encoding in PRISM. Since PTA is a subset of STS, the encoding of STS as PTA modules is a one-to-one mapping, with the syntax adapted to match the PRISM input language. All the global variables and their assertions (weights, utility and rewards) are defined outside the module definition of the exercise agent. Apart from these, the exercise agent module uses a set of local variables. Variable *s* represent the state, $s = 0$ (Idle), $s = 1$ (Op), $s = 2$ (Comm), $s = 3$ (Exc_sc), $s = 4$ (Exc1), $s = 5$ (Exc2), $s = 6$ (Fail). There are variables that represent the user's calendar exercise input (*cal_exc_u1: [0..1]*), user quit *u1_quit: [0..1]*), session completion *(ss_comp: [0..1]*, where *0* indicates that the event has not occurred, whereas *1* indicates the opposite. There are also variables to represent the exercise split sessions ($exc\_split[0..2]$), 0 representing the initial value and 1 and 2 representing the two split sessions respectively, and the exercise recommendations (exc_rec [0..2]) where 0 represent the initial condition and 1 indicating that exercise category 1 is chosen and 2 indicates that category 2 is chosen. Variable *x* is a clock variable. The invariant associated with the states (Lines 15-17) depend on the component's execution time (defined at the interface of the AADL component's model). The invariant of state Op is $x \leq Exec\_time$. The transitions defined in Lines 18-27 follow the transitions definition of the Agent Annex specification of the exercise Agent. Finally, in Lines 29-33, we show the association of rewards to the respective states or transitions. After modeling

the respective PTA modules, we can perform exhaustive probabilistic verification of the model, and generate probabilistic guarantees for the satisfaction of the functional and QoS requirements listed in Section 11.3.2.

Listing 11.3: An excerpt of the PRISM Model of an Exercise Agent

```
pta
const double w1=1.0;
const double w2=1.0;
formula utility1 = w1*(u_sat)+w2*(ss_comp);
formula sys_rew1=utility1;
formula sys_rew1= sys_rew1+utility1;
module Exc_agent1
  s: [0..6] init 0;
  //states 0 -Idle, 1-Op, 2-Comm, 3-Exc_sc, 4-Ex1,
  5- Ex2, 6-Fail
  cal_exc_u1: [0..1] init 0; u1_quit: [0..1] init 0;
  ss_com: [0..1] init 0; exc_split: [0..2] init 0;
  exc_rec: [0..2] init 0;
  x: clock;
  invariant
   (s=1 => x<=2)
  endinvariant
  [1]s =0 & cal_exc_u1=1 -> 0.999:(s'=1) & (x'=0) +
  0.001:(s'=6) &(x'=0);
  [2]s=1  & x=2  -> (s'=2) & (x'=0);
  [3]s=2  & h_stat_u1=1  -> (s'=3) & (x'=0);
  [4]s=3  & u1_pref=1 & u1_dis_his=1 &exc_split =0
  -> (s'=4) & (exc_rec'=1) & (exc_split'=2) & (x'=0);
  [r1]s=4 &(x=5|u1_quit=1) & h_stat_u1=1 & exc_split
  < 2 -> (s'=3) & (exc_split'=exc_split+1) & (x'=0);
  [5] s=3 & exc_split>0 & exc_split<2 & sys_rew1>
  sys_rew2 -> (s'=3) & (exc_split'=exc_split+1) & (x'=0);
  endmodule
  rewards
  s=4 & (u1_sat=0 | u1_sat=1) &(ss_com=1|ss_com=0):
  sys_rew1;
  [r1] true : sys_rew1;
  endrewards
```

The verification results are tabulated in Table 11.2. The requirements are formulated as PCTL queries and the model-checking method is *Digital Clocks*. Since PRISM, by default, returns the value for the (single) initial state of the model while model checking, we employ *filters* to verify our properties over all states. Requirement $R1$ ensures that if a fall event occurs due to a low pulse for $user1$ (Jim), and the tracker is operational, then the tracker initiates the communication between the respective fall and pulse agents associated with user Jim (the request can be assigned to any of the agent sockets depending on availability), and the probability that one of them sends an alert to caregiver indicating that there is "fall due to low pulse" is greater than 0.999 provided that at least one of the sockets of each agent is functional. Assuming that the communication via tracker takes less time, the requirement is satisfied within 10 time units. Similarly, for $R2$, we verify for $user2$ (Mary) that in case of fire and fall events occurring simultaneously, an alert indicating both events is raised and sent within 10

time units, provided that the tracker has not failed. In case of $R3$, $R4$ and $R5$, we verify the functionality of the exercise agent serving Jim. By $R3$, we establish that the exercise session is scheduled only if the corresponding health agent indicates that the user's pulse level is normal. $R4$ indicates that the initial exercise category is chosen based on user preferences and health condition. By verifying $R5$, we show that if a high pulse deviation occurs during the exercise sub-session, a low intensity exercise is chosen in the next sub-session, irrespective of user preferences. In $R6$, we illustrate a similar function as in $R2$, but assuming that the tracker has failed. In this case, the functionality is met by direct communication between the agents, which takes more time than the communication via tracker (it is shown that this requirement is satisfied within 20 time units). Next, in $R7$, we assume a fall event of $user2$, and one failed fall agent; then, a fall alert is raised and sent to the caregiver by either one of the redundant fall agents. PRISM shows that this requirement is satisfied within 20 time units.

Table 11.2 Verification results

Req.	Query	Result
R1	$filter(forall, fall\_user1 = 1 \& pulse\_user1 \leq 50 \& tracker\_fail = 0 \rightarrow P \geq 0.999\ [F((pulse\_alert0\_u1 = 3 \mid pulse\_alert1\_u1 = 3 \mid pulse\_alert2\_u1 = 3 \mid pulse\_alert3\_u1 = 3) \& (y \leq 10) \& (fall\_fail = 0) \& (pulse\_fail = 0))]$	satisfied
R2	$filter(forall, fall\_user2 = 1 \& fire\_user2 = 1 \& tracker\_fail = 0 \rightarrow P \geq 0.999[F((firefall\_alert0\_u2 = 2 \mid firefall\_alert1\_u2 = 2 \mid firefall\_alert2\_u2 = 2 \mid firefall\_alert3\_u2 = 2) \& (y \leq 10) \& (fall\_fail = 0) \& (fire\_fail = 0))]$	satisfied
R3	$filter(forall, cal\_notexc\_user1 = 1 \& tracker\_fail = 0 \& (pulse\_user1 \geq 60 \& pulse\_user1 \leq 120) \rightarrow P \geq 0.999\ [F(exc\_sch\_u1 = 1))]$	satisfied
R4	$filter(forall, exc\_sch\_u1 = 1 \& u1\_disease\_history = 1 \& u1\_pref = 2 \rightarrow P \geq 0.999\ [F(exc\_u1\_int1 = 2)]$	satisfied
R5	$filter(forall, exc\_sch\_u1 = 1 \& interval = 1\ \& y \leq 5 \& pulse\_user1 \geq 200 \rightarrow P \geq 0.999\ [F(exc\_u1\_int2 = 1)]$	satisfied
R6	$filter(forall, fall\_user2 = 1 \& fire\_user2 = 1 \& tracker\_fail = 1 \rightarrow P \geq 0.999\ [F((fall\_alert0\_u2 = 2 \mid fall\_alert1\_u2 = 2 \mid fall\_alert2\_u2 = 2 \mid fall\_alert3\_u2 = 2) \& (y \leq 20) \& (fall\_fail = 0) \& (fire\_fail = 0))]$	satisfied
R7	$filter(forall, fall\_user2 = 1 \& tracker\_fail = 0 \& fail1\_fall = 1 \& fall2\_fall = 0 \rightarrow P \geq 0.999\ [F((fall\_alert2\_u2 = 1 \mid fall\_alert3\_u2 = 1) \& y \leq 20)]$	satisfied

11.7 Related Work

The latest research in AAL systems has shown considerable progress in order to meet the safety and day-to-day requirements of the growing elderly population in the society [13]. Modern AAL systems are designed to tackle numerous functions, and to cater for multiple, possibly distributed users, which makes the system design more complex, and calls for design-time formal analysis.

Some related work is directed towards providing formalisms for agents in terms of various logics [14, 15]. However, some others have proceeded further to develop specification languages/methodologies for agent systems. Some examples include CASL [16], DESCARTES [17], etc. These methodologies employ different formalisms, however some of them are complex and are not expressive enough, like in case of CASL. For DESCARTES, tool support for executing the specifications is also provided. Although the approach is promising, the DESCARTES language is still missing constructs to specify adaptive capabilities of agents, nor it provides an analysis framework for MAS. One of the other common approaches, popular in industry also, is the Agent UML [18] one. The approach does not specify the architectural constructs of the system, and lacks formal analysis, unlike the framework that we present in this paper. Few works have considered the specification and formal analysis of agent behavior in architecture description languages [19]. The AADL-based modeling framework for multi-agent systems, which we propose in this paper, has the benefit of being integrated into a popular framework that also provides tool support.

There are also some approaches that focus on the formal verification of AAL systems. An interesting related work is that of Rodrigues et al. [2], who perform dependability analysis of AAL architectures using UML and PRISM. Other interesting research work uses temporal reasoning [20] to formally verify the reliability of AAL systems. However, the above focus only on QoS requirements, and do not look into the critical functions of AAL systems, which require decision making. Unlike these approaches, we carry out our analysis on an agent-based AAL system architecture, focusing on both functional and QoS requirements, and propose a complete modeling and verification framework for distributed AAL systems that involve real-time, fault-tolerant and probabilistic behaviours. As an advantage if compared to another work [1], the verification results obtained with PRISM are exhaustive. In the mentioned work, the authors have proposed a formal assurance framework for AAL system architectures described in AADL, and showed how to verify them in UPPAAL SMC. As different from the work in this paper, the approach assumes a centralized system architecture, and the only probabilistic behaviour considered in the system is component failure, which can occur arbitrarily. In addition, the statistical analysis with UPPAAL SMC, is not exhaustive, but it relies instead on a finite number of simulations.

11.8 Discussion

This paper presents an architecture for MAS, which we model in the architecture language AADL that we extend with an agent annex intended to model real-time, fault-tolerant and probabilistic behavior of agents, in a unified manner. This approach allows an agent to synchronize only with a limited number of agents in the system (according to its dependency), unlike the traditional case where each agent has to communicate with every other agent in the system to achieve a consistent view of the environment before making a decision [11]. We show the design of our MAS architecture applied to AAL domain with 2 agent categories- simple reflex agents, that use *if-then-else* rules and complex intelligent agents that employ learning techniques, like *Reinforcement Learning*. Although we have demonstrated the use-case of Ambient Assisted Living in the paper, the approach fits well for any other applications employing MAS for handling multiple safety critical applications in real-time, e.g., those of automotive systems for which earlier stage analysis is beneficial.

The modeling framework used in this paper is relying on the Architecture Analysis and Design Language (AADL), one of the best-suited architecture description languages to describe real-time embedded systems [5]. Although MAS specifications based on logics and domain specific languages do exist and are popular, they are mostly limited to specification of properties at the agent level and also do not have tool support (see Section 11.7). AADL, on the other hand, allows us to focus on the component level (here *agents*) and also at the system level (*MAS architecture*) and can effectively model agents' real-time characteristics. With our proposed extension to AADL using Agent Annex, a user can also specify the intelligent agent behaviours (which are often probabilistic) and their failures. AADL also supports a graphical plug-in in OSATE tool to visualizethe model and supports analysis with respect to latency, schedulability, resource utilization, etc. [21].

In this work, we encode the semantics of the AADL model and its agent annex as Stochastic Transition Systems. The encoding is suitable due to the probabilistic behaviour of such systems and allows it to be model-checked exhaustively by probabilistic model checkers, like PRISM, or statistically by simulation-based model checkers like UPPAAL SMC. This paper shows a reduced and abstract architecture with only 4 types of agents and hence can be verified exhaustively with PRISM.

11.9 Conclusions and Future Work

In this paper, we have proposed an architecture modeling and formal analysis framework for agent-based AAL systems characterized by intelligent, probabilistic, and real-time behaviours. The intelligence is incorporated by using learning algorithms, in our case, the Reinforcement Learning algorithm. The modeling framework is based on one of the well-established architecture description languages for modeling real-time em-

bedded systems, called AADL. As the core AADL does not suffice to represent the probabilistic and non-deterministic behavior of our system, we propose an annex extension to AADL, the so-called Agent Annex that we formally encode as a stochastic transition system. In order to verify a set of critical functional and QoS requirements like timeliness, fault-tolerance etc., we use an exhaustive probabilistic model-checking method, using the state-of-art model checker PRISM.

Our contribution paves the way for the development of formally assured distributed, adaptable, scalable, fault-tolerant systems, with intelligent behaviours and autonomy. The scalability of the proposed framework is supported by the semantic definition of AADL elements that allows an encoding in UPPAAL SMC for instance, for statistical model checking of models that exceed the boundaries of exhaustive model checking. As future work, we intend to extend our architecture with multiple categories of agents and integrate the Agent Annex to the core AADL.

Acknowledgements

This work is supported by the EU Celtic Plus /Vinnova project, Health^{5G}- Future eHealth powered by 5G, which is gratefully acknowledged.

Bibliography

[1] Ashalatha Kunnappilly et al. Assuring intelligent ambient assisted living solutions by statistical model checking. In *International Symposium on Leveraging Applications of Formal Methods*, pages 457–476. Springer, 2018.

[2] Genaína Nunes Rodrigues, Vander Alves, Renato Silveira, and Luiz A Laranjeira. Dependability analysis in the ambient assisted living domain: An exploratory case study. *Journal of Systems and Software*, 85(1):112–131, 2012.

[3] Dante I Tapia et al. An ambient intelligence based multi-agent system for alzheimer health care. *International Journal of Ambient Computing and Intelligence*, 1(1):15–26, 2009.

[4] Richard S Sutton, Andrew G Barto, et al. Introduction to reinforcement learning. 135, 1998.

[5] Peter H Feiler et al. The architecture analysis & design language (AADL): An introduction. Technical report, Carnegie-Mellon Univ Software Engineering Inst, 2006.

[6] P Dissaux, Jean-Paul Bodeveix, M Filali, P Gaufillet, and F Vernadat. Aadl behavioral annex. In *Proceedings of DASIA conference, Berlin*, volume 32, 2006.

[7] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 200–204. Springer, 2002.

[8] Luca De Alfaro. Stochastic transition systems. In *International Conference on Concurrency Theory*, pages 423–438. Springer, 1998.

[9] Gethin Norman et al. Model checking for probabilistic timed automata. *Formal Methods in System Design*, 43(2):164–190, 2013.

[10] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5):512–535, 1994.

[11] Ashalatha Kunnappilly, Alexandru Sorici, Imad Alex Awada, Irina Mocanu, Cristina Seceleanu, and Adina Madga Florea. A Novel Integrated Architecture for

Ambient Assisted Living Systems. In *Computer Software and Applications Conference (COMPSAC), 2017 IEEE 41st Annual*, volume 1, pages 465–472. IEEE, 2017.

[12] PA USA Society of Automotive Engineers, Warrendale. AE-AS5506/1, SAE Architecture Analysis and Design Language (AADL) Annex Volume 1, Annex C: AADL Meta-Model and Interchange Formats, 2006.

[13] Parisa Rashidi and Alex Mihailidis. A survey on ambient-assisted living tools for older adults. *IEEE journal of biomedical and health informatics*, 17(3):579–590, 2013.

[14] Hai-yan Che et al. A Description Logic Method of Formalizing the Specification of Multi-Agent System. In *Machine Learning and Cybernetics, 2006 International Conference on*, pages 61–65. IEEE, 2006.

[15] Jiewen Luo et al. Multi-agent cooperation: A description logic view. In *Pacific Rim International Workshop on Multi-Agents*, pages 365–379. Springer, 2005.

[16] Steven Shapiro, Yves Lespérance, and Hector J Levesque. The cognitive agents specification language and verification environment for multiagent systems. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*, pages 19–26. ACM, 2002.

[17] Michael A Medina et al. *An approach to deriving reactive agent designs from extensions to the Descartes specification language*. IEEE, 2007.

[18] Bernhard Bauer, Jörg P Müller, and James Odell. Agent UML: A formalism for specifying multiagent software systems. *International journal of software engineering and knowledge engineering*, 11(03):207–230, 2001.

[19] Flavio Oquendo. π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–14, 2004.

[20] using temporal logic and model checking in automated recognition of human activities for ambient-assisted living.

[21] *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language*.

Paper E

Chapter 12

Paper E: An End-User Perspective on the CAMI Ambient And Assisted Living Project

Imad Alex Awada, Oana Cramariuc, Irina Mocanu, Cristina Seceleanu, Ashalatha Kunnappilly, Adina Magda Florea.
In Proceedings of the 12th Annual International Technology, Education and Development Conference (INTED), March 2018, Valencia, Spain.

Abstract

In this paper, we present the outcomes and conclusions obtained by involving seniors from three countries (Denmark, Poland and Romania) in an innovative project funded under the European Ambient Assisted Living (ALL) program. CAMI stands for " Companion with Autonomously Mobile Interface " in " Artificially intelligent ecosystem for self-management and sustainable quality of life in AAL ". The CAMI solution enables flexible, scalable and individualised services that support elderly to self-manage their daily life and prolong their involvement in the society (sharing knowledge, continue working, etc). This also allows their informal caregivers (family and friends) to continue working and participating in society while caring for their loved ones. The solution is designed as an innovative architecture that allows for individualized, intelligent self-management which can be tailored to an individual's preferences and needs. A user-centred approach has ranked health monitoring, computer supervised physical exercises and voice based interaction among the top favoured CAMI functionalities. Respondents from three countries (Poland, Romania and Denmark) participated in a multinational survey and a conjoint analysis study.

12.1 Introduction

In the context of unprecedented worldwide demographic changes, information and communication technologies (ICT) are increasingly sought for their potential to help aging adults and seniors to live independently in their home environment. While innovation in this field is rapidly picking up, the full impact of such technologies can be attained only through a wide spread adoption of such technologies by the elderly population. Consequently, several initiatives at both national and cross-national level are actively supporting the development of AAL ICT through a user-centered approach that is expected to increase acceptance and reduce learning barriers. In this paper, we present the outcomes and conclusions obtained by involving seniors from three countries (Denmark, Poland and Romania) in a project funded under the European Ambient Assisted Living (ALL) program. CAMI stands for " Companion with Autonomously Mobile Interface" in " Artificially intelligent ecosystem for self-management and sustainable quality of life in AAL"[1]. The project consortium comprises eight SME's and universities from five European countries which are developing a fully integrated AAL solution at the overlap of tele- care and health, smart homes and robotics (see Figure 12.1). The CAMI solution enables flexible, scalable and individualized services that support elderly to self-manage their daily life and prolong their involvement in the society (sharing knowledge, continue working, etc) while allowing their informal caregivers (family and friends) to continue working and participating in society whilst caring for their loved ones. The solution is designed as an innovative architecture that allows for individualized, intelligent self-management which can be tailored to an individual's preferences, culture, level of comprehension, skill, educational needs and learning style. A combination of advanced reasoning algorithms with a voice-based interface is easing the self-management and decision-making process. The decision support module is an expert system that acts as the central integration point for CAMI. It is a dockerized container, deployed in the cloud. A user-centred approach has ranked health monitoring, computer supervised physical exercises and voice based interaction among the top favoured CAMI functionalities. Respondents from three countries (Poland, Romania and Denmark) participated in a multinational survey and a conjoint analysis study. E-health solutions were perceived as very useful and 59% of the respondents considered the graphic display of various health measurements (e.g. blood pressure, heart rate, oxygen levels) as an interesting feature. The ability to share health measurements with various doctors was perceived as useful by 60% of the respondents. Computer supervised physical exercises ranked third in the conjoint analysis. All the voice control functionalities received more than 50% votes which classifies them as promising features. The physical exercise module is based on exergames with two avatars: the training avatar and the avatar of the user. The training avatar is performing different physical exercises which have to be reproduced by the user. The user's movements are compared with the movements of the avatar based on algorithms for comparing two nonlinear series. At the end of the exercise, the user receives a score that reflects the

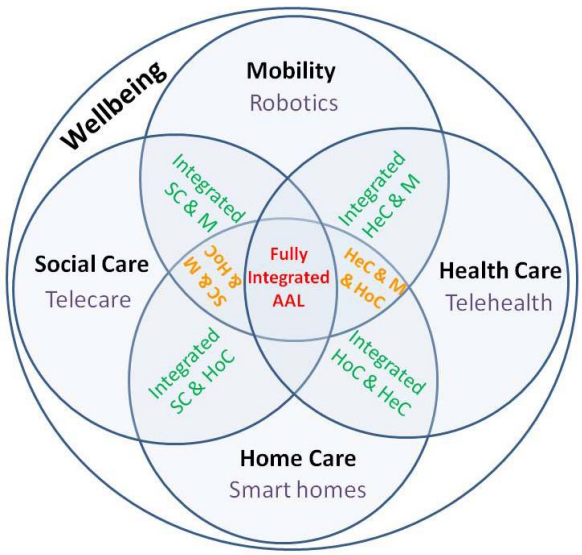


Figure 12.1: Venn diagram of a fully integrated AAL solution

correctness of the performed exercises. The results are saved in a database, such that they can be analysed by a caregiver or by a medical specialist. The voice interface is composed of five main parts: Automatic Speech Recognition, Natural Language Understanding, Dialog Management module, Natural Language Generation, and Text to Speech synthesis. Since, the user should always be able to interact with the CAMI platform, regardless of the status of the internet connection, the vocal interface should work also offline ensuring basic system functionalities. Therefore, the vocal interface will have two working modes: (1) an online mode that depends on internet connectivity and (2) an offline mode (a limited version) that doesn't depend on internet connection. A user-centred approach has ranked health monitoring, computer supervised physical exercises and voice based interaction among the top favoured CAMI functionalities. Respondents from three countries (Poland, Romania and Denmark) participated in a multinational survey and a conjoint analysis study. E-health solutions were perceived as very useful and 59% of the respondents considered the graphic display of various health measurements (e.g. blood pressure, heart rate, oxygen levels) as an interesting feature. The ability to share health measurements with various doctors was perceived as useful by 60% of the respondents. Computer supervised physical exercises ranked third in the conjoint analysis. All the voice control functionalities received more than 50% votes which classifies them as promising features. The physical exercise module is based on exergames with two avatars: the training avatar and the avatar of the user. The

training avatar is performing different physical exercises which have to be reproduced by the user. The user's movements are compared with the movements of the avatar based on algorithms for comparing two nonlinear series. At the end of the exercise, the user receives a score that reflects the correctness of the performed exercises. The results are saved in a database, such that they can be analysed by a caregiver or by a medical specialist. The voice interface is composed of five main parts: Automatic Speech Recognition, Natural Language Understanding, Dialog Management module, Natural Language Generation, and Text to Speech synthesis. Since, the user should always be able to interact with the CAMI platform, regardless of the status of the internet connection, the vocal interface should work also offline ensuring basic system functionalities. Therefore, the vocal interface will have two working modes: (1) an online mode that depends on internet connectivity and (2) an offline mode (a limited version) that do not depend on internet connection.

12.2 An Overview of the CAMI Platform Architecture

The architecture of the CAMI system is developed aiming at the seamless integration of various assisted-living functionalities, like health monitoring, fall detection, home monitoring, robotic platform support etc., ensuring modularity and re-use. A detailed description of CAMI architecture was presented in our previous work [2] [3]. In this paper, we present a smaller implemented version of the system architecture as shown in Figure 12.2. The integration of various functionalities is achieved by three main modules: a) CAMI Gateway (running on the SNG-Gateway developed by the Eclexys Sagl partner, b) CAMI Cloud, and c) CAMI multi-modal user interface (e.g. the 3rd party health platforms (Linkwatch [4] and OpenTele [5] and vocal interaction with the CAMI Cloud). The CAMI Gateway connects with a multitude of Bluetooth and Z-Wave compatible health measurement and home monitoring sensors (e.g. the A&D UA-651 BLE blood pressure meter, the Onyx II Model 9560 oxymeter, Fibaro Temperature and Motion Sensor FGMS-001, Z-Wave 3 in 1 Sensor (temperature, illumination, door) PHI_PSM01), Vibby fall detection sensor, etc. The CAMI Cloud allows the system to perform two other essential integrations:

- From the input perspective: the CAMI Cloud allows access to information from sensors that publish their data directly to the cloud service (e.g. Fitbit bracelet, WiFi weight scale, smartwatches etc.)
- From an output/sharing perspective: the CAMI Cloud allows dissemination/replication of data to other health monitoring platforms (e.g. Linkwatch, OpenTele). This type of integration allows end-users to monitor the health parameters collected through the CAMI system via web-accessible graphical interfaces.

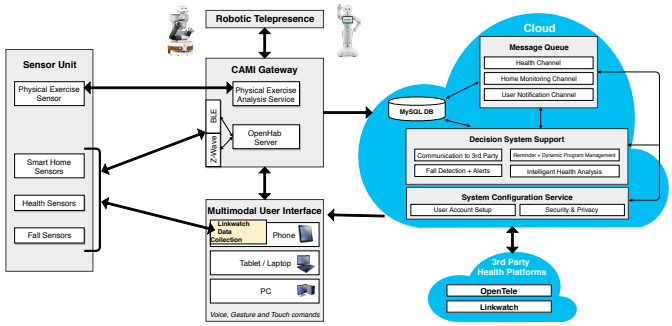


Figure 12.2: CAMI architecture block diagram

The CAMI Multi-Modal Interface allows for user interaction with the system. This is achieved by smartphone application (implemented on iOS) and the CAMI Linkwatch web interface, and vocal interaction. For each integration aspect, a dedicated micro service running in its own Docker container or using its own RESTful API is implemented.

12.3 Results

12.3.1 The CAMI end-user perspective

A total of 105 respondents (55 to 75 years old) from Romania (42), Poland (37) and Denmark (26) have participated in the multinational survey. These are CAMI's primary end-users, that is, older adults and elderly who have benefited from the digital revolution and are therefore expected to have an increased acceptance of innovative ICT solutions. Out of the respondents 49 were males and 56 females living in urban and sub-urban areas (87%). Most of the respondents (46%) had a master degree or higher while 28% held a post-secondary school qualification. Most of the respondents were married or in partnership (64%), and almost 30% of them were living alone being widowed or separated. More than a half of the respondents were retired (55%), while the rest were still active (employed or running their own business). All respondents were informed about the anonymity of the survey and have provided an informed consent for their participation.

Several of the CAMI technologies were already used by the respondents as independent devices. For example, 85% of the respondents have 4 or less mobile devices (smartphone, tablet, etc). The median value of number of possessed devices for Denmark is 4, while for Poland and Romania is 3. The number of possessed devices does not depend on gender and age. Small influence has the level of education and number of people the respondents live with. In addition, respondents expressed their interest in

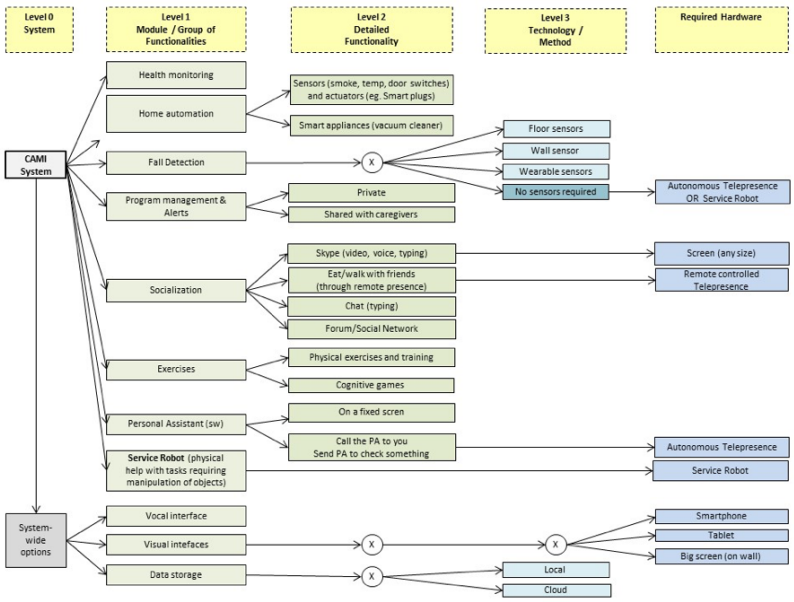


Figure 12.3: Summary sketch of CAMI's envisioned functionalities distributed hierarchically.

CAMI's technologies. E-health solutions were well perceived as very useful and 59% of the respondents considered the graphic display of various health measurements (e.g. blood pressure, heart rate, oxygen levels) as an interesting feature. The ability to share health measurements with various doctors was perceived as useful by 60% of the respondents. Physical and mental exercises were of interest for 41% of the respondents. All the voice control functionalities received more than 50% votes which classifies them as promising features. Nevertheless, more than half (60%) considered that a mobile device is an acceptable interface with the system. Robotic platforms were perceived as being useful for a number of tasks such as house supervision (51%), telepresence (54%), manipulation of objects (50%), etc. A total of 87% of the respondents were worried about the price of the CAMI solution. Consequently, a renting scheme was more appealing to the participants. They were interested to own the health monitoring devices and rather rent other less personal devices and sensors such as those for home monitoring, serious games, robotic platforms, etc. With the multinational survey outlined above establishing the interest of the CAMI primary users for the targeted technologies, we proceeded to rank these technologies according to the user's preference. For this purpose, we employed the best-worst scaling (BWS) method for a total of 57 elderly

participants: 25 Romanians (11 female, 14 male), 20 Polish (13 female, 7 male), and 12 Danish (6 female, 6 male) respondents. The BWS belongs to the so-called 'stated preferences' family of method and was first proposed by Louviere in a series of articles, as an alternative to classical discrete choice experiments [6]. In BWS, respondents were not asked how much they prefer certain alternatives of the CAMI solutions compared with each other, but only to choose which options they prefer and which they donâĂŹt. A list of 22 functionalities was defined as bases for the survey's combinations pool (see Figure 12.3).

All relevant functionalities were subject to a randomization algorithm utilized by the BWS method, whose purpose was to maximize the probability of appearance of each functionality in a certain choice set. The BWS calculated scores are directly comparable in terms of strength or magnitude of preferences. More specifically, a functionality with the computed preference weight of 8 is twice as preferred as a functionality with the computed preference weight of 4. In our research, the two most preferred functionalities (1 and 2) were twice as preferred as more than half of all preferences (for instance, 4, 18, 12, 10, 17 and so on). As depicted in Figure 12.4, functionalities 1 (basic health parameters monitoring), 2 (smart house with various sensors, such as smoke, temperature, open doors, etc.) and 14 (computer supervised physical exercise and training program) were ranked the three most preferred, whereas functionalities 4 (fall detection alert-able floor), 13 (socialization via forums), and 22 (have the system-acquired data stored in the cloud) were ranked least preferred. Further discussions within focus groups organized in all three countries, i.e. Romania, Poland and Denmark, have revealed that fall detection was ranked low only in implementations requiring substantial changes of the home environment and high investments (alert-able floor). On the contrary, fall detection through wearable sensors was considered an essential feature, especially by the caregivers. The focus groups were conducted with 5-8 elderly and with a large variety of secondary stakeholders such as nurses, IT specialists, insurance companies, etc. The vocal interface feature, which was greatly overlooked during both the multinational survey and the BSW analysis was appreciated during the focus groups and subsequent demonstrations performed during the CAMI project. In the next sections, we are presenting the implementation of some of the CAMI technologies which were ranked top by the users participating in the project.

12.3.2 Health monitoring and fall detection

The health monitoring functionality in CAMI is allowing users to perform scheduled (e.g. according events scheduled in the user's calendar) monitoring of important health parameters, i.e. blood pressure, heart rate, blood glucose, weight, blood oxygenation. The integrated medical devices are transferring the acquired data via Bluetooth to the CAMI Gateway and then further to the CAMI database. The acquired data is checked against the user's profile to identify important deviations from acceptable limits. Deviations trigger alert message to the user and the caregiver. The stored data can be used to

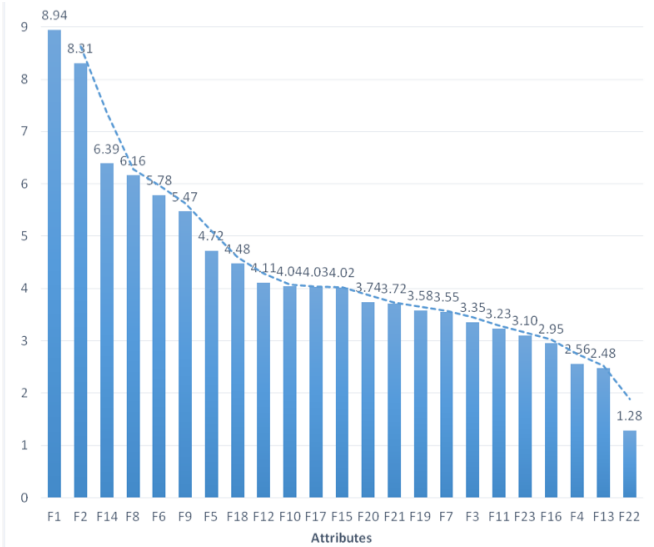


Figure 12.4: Preferences hierarchy of all functionalities for all respondents. Values represent MaxDiff scores.

plot the evolution in time of the physiological parameters. Normal or acceptable limits for each of the parameters will be plotted together with the recorded data. Falls are serious threats to elderly people living alone and can even result in life threatening situations when the fall is critical and not addressed within a specific time. As a result, in CAMI, the fall detection functionality together with alarm generation on the occurrence of a serious fall, needed in order to automatically inform caregivers and family is given due importance. To detect a fall, we employ the Vibby Oak fall detection sensor, along with its IoT gateway, Vibby Leaf (both produced by Vitalbase). Vibby Oak is a wearable sensor that can be worn around the wrist like a watch, or as a pendant around the neck. They are designed to automatically detect dangerous (heavy) falls of the wearer who lies on the floor. The device also has a manual trigger to push the alarm off, if the user has recovered successfully. A dangerous fall is detected if: a) a body in standing position is followed by: b) a quick and sudden loss of gravity or verticality followed by: c) a sudden and strong impact on the floor, and d) a lying position on the floor with or without activity. If these four phases have occurred, an automatic alarm is then activated. The Vibby Oak algorithm reduces the false fall alarms significantly, although it does not completely remove them. The Vibby Leaf gateway is an ISM 868Mhz transceiver designed to connect to the Vibby Oak fall detector, with data received by ISM-band radio communication. To integrate it with the CAMI gateway, we forward the fall event

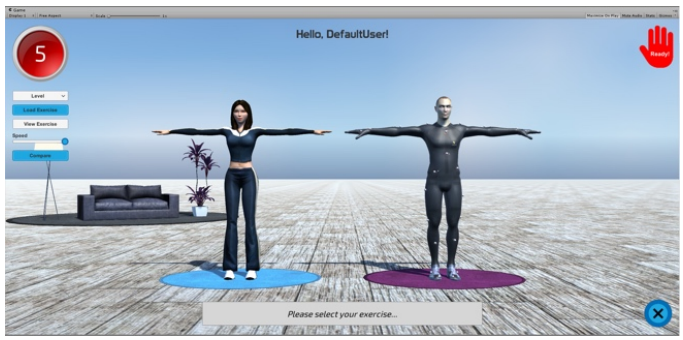


Figure 12.5: Screenshot of the application.

to the CAMI gateway, from where it gets pushed to the Decision Support System (DSS) placed in the CAMI cloud, which generates a notification for the caregiver regarding the fall.

12.3.3 Computer supervised physical exercises

This functionality aims to (1) improve the user's lifestyle by providing regular physical activity that is consistent with his / her medical condition; (2) monitor the user's activity and provide motivational feedback; provide a personalized exercise program that can be adapted by a medical specialist, depending on patient progress. The application is developed using the Unity 3D engine and is based on two avatars: the training avatar and the avatar of the user. The training avatar is performing different physical exercises which the user must reproduce. The user's movements are compared with the movements of the avatar based on algorithms for comparing two nonlinear series. The results are saved in a database, such that they can be analyzed also at later times. At the end of the exercise, the user is receiving a score reflecting the correctness of the performed exercises. A screen shot of the application, with both avatars, is given in Figure 12.5.

The Kinect v2 sensor is used to map the user's movements to its avatar. The sensor provides 25 joints for a user at approximately 30 frames per second. For the implemented exercises, only 20 joints are used. For each joint we compute a 3D rotation using quaternions relative to the parent bone. The application aims to personalize the exercises according to the medical or physical condition of each user. For this purpose, we associated weights with each joint in order to reflect the user's condition. Each weight has a value in between 0 and 1 which can be personalized for each user. In order to compare the trainer and user movements, the similarity between the set of quaternion values for each joint, computed for each frame, is compared using the Dynamic Time Warping algorithm (DTW) [7]. DTW computes the similarity between two series by

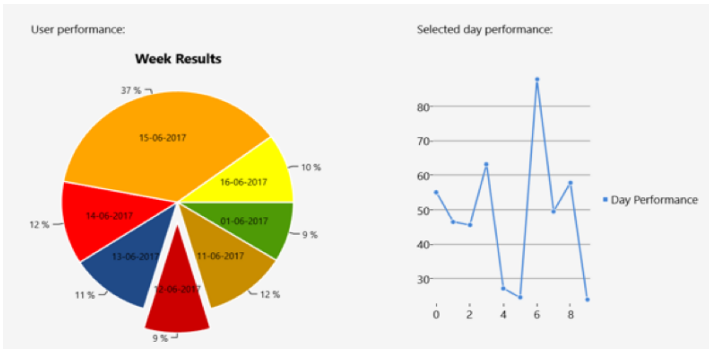


Figure 12.6: Scores obtained by one user.

calculating the minimum distance between them. The Kinect sensor is introducing input noise when collecting the user's movements. Hence, the result obtained by applying the DTW algorithm to two sets of frames must be processed in order to obtain a robust result that reflects the correctness of the user's movements. The score provided to the user, reflecting his results is computed using the equation 12.1:

$$Similarity(f_{\text{ref}}, f_{\text{user}}) = 1 - DTW/ref_{\text{MAX}} \tag{12.1}$$

where DTW is the distance between f_{ref} and f_{user} computed using DTW; fref and fuser are are the set of frames associated to the trainer and to the user; ref_{MAX} represents the highest value obtained by applying the DTW algorithm. The latter is obtained experimentally based on the user's performance. The similarity between joints is computed with DTW using the Euclidean distance. Values obtained for joints similarities are normalized in the range [0, 1] by dividing to the maximum obtained value. If one value is greater than a fixed threshold (experimentally obtained) then the movement wasn't performed correctly based on that joint. We experimented with different distance metrics for computing the DTW, i.e. inner product, euclidean distance, squared Euclidean distance and Manhattan distance. Four cases were considered: a) M1 - user movements are performed similar with the reference exercise; b) M2 - user movements are performed slower than the reference exercise; c) M3 - user movements are partially wrong than the reference one and d) M4 - user movements are slower and different than the reference one. The similarity computed with the squared Euclidean distance and the inner product distance can differentiate between these 4 type of cases. The other two considered methods are not able to differentiate between M2 and M3. The user can visualize the results obtained during a week, as shown in Figure 12.7. One day exercises can be also selected for visualization.

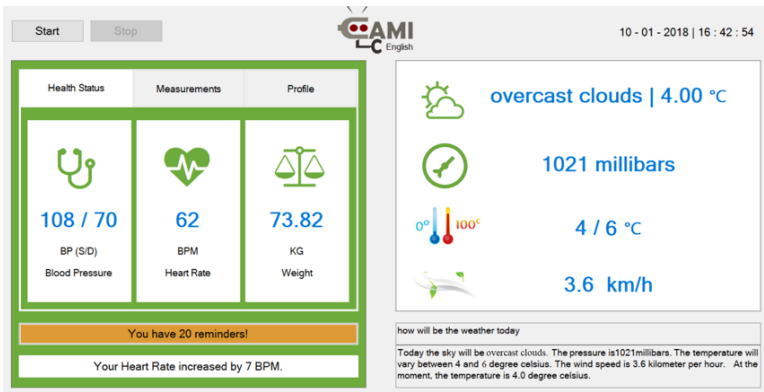


Figure 12.7: Scores obtained by one user.

12.3.4 CAMI Vocal Interface

The vocal interface integrated in CAMI allows oral (speech and not speech) interactions between the user and the system (see Figure 12.7). After analyzing existing solutions in the field, we developed a vocal interface that is composed of five main modules: Automatic Speech Recognition (ASR), Natural Language Understanding (NLU), Dialog Management (DM), Natural Language Generation (NLG), and Text to Speech synthesis (TTS). Since, the user should always be able to interact with the CAMI platform, regardless of the status of the internet connection, the vocal interface should work also offline ensuring basic system functionalities. Therefore, the vocal interface has two working mode: one that depends on the internet connectivity that will be used normally (online mode) and the other that doesn't depend on internet connection that will be used in case of connection lost (offline / limited mode).

Briefly, ASR is the technology that allows a computer to identify the words spoken by a person into a microphone and to convert them to writing text. We use the Windows Speech Technology as our ASR solution. It offers a basic speech recognition infrastructure that digitizes the acoustic signals and recovers words and speech elements from the audio input. To access and extend this basic speech recognition technology, we use the "System.Speech.Recognition namespace" and define algorithms for identifying and acting on specific phrases or word patterns. We also manage the runtime behavior of this infrastructure and we created a grammar that consist of a set of rules and constraints to define words and phrases that will be recognized as meaningful input. The NLU module aims to extract the semantic meaning from the text received from the ASR module by converting the received words to a machine-reading representation. This module is also responsible to correct any errors made by the ASR module. We use the language Understanding Intelligent Service (LUIS) from Microsoft as our NLU solution. By sending

Table 12.1 Results of the ASR module performance tests.

User's input	Score Quiet Env	Score Noisy Env
Who are you	98.51	94.63
How will be the weather today	98.74	95.72
What I have scheduled for today	98.05	94.62
What is my health status	99.73	95.70
Call Bogdan	99.89	96.12
How much have I walked	97.15	94.21
Show my calendar	98.56	94.81
Take a new health measurement	98.16	94.31
I forgot how to use the device	99.15	94.29
Display blood sugar measurement	98.89	94.58

the resulted text from the ASR module to LUIS, the system will receive back relevant, detailed information regarding the user's request. For each domain of interaction, we designed its proper domain-specific language model and tailored it to the need of the system. The DM module is responsible for the state and flow of the conversation. It receives as input some relevant information regarding the user's request that were generated by the NLU. The output of this module is a semantic representation of a list of instructions of the dialog system that determines which should be the system's answer in response to the user's processed input. The NLG module is the natural language processing task of generating natural language from a machine representation system (such as a knowledge base or a logical form). It may be viewed as the opposite of the NLU. We use the Microsoft Bot Framework from Microsoft as a solution that combines the DM and NLG modules, it is a platform that allows to build, connect, test, and deploy powerful and intelligent bots. We built the bots from scratch, using the Bot Builder SDK for .NET and Node.js provided by the Microsoft Bot Framework. The TTS module is responsible to artificially produce any generated output normal language text into human speech that will be heard over the speakers of the system. The quality of a speech synthesizer is judged by its similarity to the human voice and by its ability to be understood clearly. We use the SpeechSynthesizer Class included in the Windows Speech Technology. The implementation outlined above is the way in which the modules of the voice interface work in the online mode. Regarding the offline mode of the voice interface there is no change at the levels of ASR and TTS modules with the exception of the number of the recognized commands which are reduced to those that ensure the basic system functionalities. Regarding the other three modules, we developed an algorithm in which the NLU compares its generated output with an array that contains all the commands that should be recognized. If no match is found, the system will tell

Table 12.2 Results of TSR module performance testing.

System spoken out	Un.QE	Un. NE	Cl.QE	Cl.NE
I am CAMI, your smart personal assistant How can I help you?	10.00	9.75	9.50	9.50
Here is the home control main page.	10.00	9.50	9.50	9.50
What health measurement do you want to take ?	9.75	8.75	9.00	9.00
Here is your weekly calendar. Your time is getting filled.	10.00	9.75	9.50	9.50
Here are your blood sugar variations during past month.	9.75	8.75	9.25	9.25

the user that the command is not recognized, and will ask for a new one. If a match is found, the system will execute the predefined content and will generate the outputs that correspond to the match (local variable involved only). We tested our voice interface in the laboratory. The tests have been done on a set of 200 interactions between 5 different users and the machine. Each user repeated 20 interactions with the system 2 times using a Plantronics Voyager 5200 UC Microphone integrated into the system in a quiet then noisy environment. We tested the solution on an HP ZBook 15 G3 (Core i7 2.60 Ghz, 8 GB RAM, integrated stereo speakers), having Windows 10 - 64 bits operating system. For the ASR module we used the Levenshtein Distance to calculate the differences between the speech recognition results and the original texts [8]. Some of the results are listed in Table 12.1.

For the TTS module we used a questionnaire to find out if the user understood the spoken output and its degree of the clearness. For each question, the user answered with a score from 0 to 10 where 10 represents represent full understanding or excellent clearness. Results are listed in Table 12.2. Column 1 describes the system spoken output, column 2 and 3 specify the user satisfaction in understanding in quiet and noisy environment respectively, and column 3 and 4 represent the system satisfaction in clearness in quiet and noisy environment respectively.

The obtained results are satisfying and are considerably improved when using an advanced microphone. Furthermore, we expect also the TTS module to exhibit improved results when using high-quality speakers.

12.4 Conclusions

A user centered design involving participants from Denmark, Poland and Romania was employed for the development of the CAMI platform in order to best fit the requirements and needs of elderly people. In addition to a multinational survey, the BWS method was used to obtain not only a general opinion of the elderly people on the CAMI solution but also a ranking of the CAMI functionalities. Functionalities related to health monitoring and prevention, including fall detection and computer supervised exercises, are highly ranked by the elderly users. Further discussions within focus groups and demonstration sessions organized in three countries also reveal vocal interaction with the CAMI platform is considered important when elderly people are involved. The obtained results helped us propose guidelines for decreasing acceptance barriers of ICT solutions among the aging population. Moreover, we implemented these in the design and development of CAMI's functionalities. Through the integration of e-health application for self-monitoring of health parameters (e.g. blood pressure, glucose, heart rate, weight, etc.) with computer supervised physical exercise the CAMI end-users can learn how to for maintaining a healthy and independent lifestyle. In addition, fall detection and alerting is increasing their level of security and confidence. Extensive field trials are planned to gather user's assessment of the implementation of the CAMI functionalities described in this paper.

Acknowledgement

This work was supported by the Active and Assistive Living (AAL) program through a grant of the Romanian National Authority for Scientific Research and Innovation, "CAMI- The Artificially intelligent ecosystem for self-management and sustainable quality of life in AAL", project number AAL-2014-1-087.

Bibliography

[1] Active and Assisted Living Programme: CAMI. www.camiproject.eu. Accessed: 2018-01-15.

[2] Ashalatha Kunnappilly, Alexandru Sorici, Imad Alex Awada, Irina Mocanu, Cristina Seceleanu, and Adina Madga Florea. A Novel Integrated Architecture for Ambient Assisted Living Systems. In *Computer Software and Applications Conference (COMPSAC), 2017 IEEE 41st Annual*, volume 1, pages 465–472. IEEE, 2017.

[3] Alexandru Sorici, Imad Alex Awada, Ashalatha Kunnappilly, Irina Mocanu, Oana Cramariuc, Lukasz Malicki, Cristina Seceleanu, and A Florea. CAMI-An Integrated Architecture Solution for Improving Quality of Life of the Elderly. In *International Conference on IoT Technologies for HealthCare*, pages 141–144. Springer, 2016.

[4] Linkwatch. https://www.linkwatch.se. Accessed: 2018-01-15.

[5] Opentele. https://www.opentelehealth.com. Accessed: 2018-01-15.

[6] JL Louviere. Conjoint Analysis. *Advanced methods of marketing research*, pages 223–259, 1994.

[7] Hiroaki Sakoe and Seibi Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE transactions on acoustics, speech, and signal processing*, 26(1):43–49, 1978.

[8] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

MÄLARDALEN UNIVERSITY
SWEDEN

Address: P.O. Box 883, SE-721 23 Västerås. Sweden
Address: P.O. Box 325, SE-631 05 Eskilstuna. Sweden
E-mail: info@mdh.se **Web:** www.mdh.se

ISBN 978-91-7485-425-1
ISSN 1651-9256