sensors

MDPI

Article

A Model-Checking-Based Framework for Analyzing Ambient Assisted Living Solutions †

Ashalatha Kunnappilly [1,*], Raluca Marinescu [2] and Cristina Seceleanu [1]

1 School of Innovation, Design and Technology, Mälardalen University, 72220 Västerås, Sweden; cristina.seceleanu@mdh.se
2 Bombardier Transportation, 72223 Västerås, Sweden; raluca.marinescu@rail.bombardier.com
* Correspondence: ashalatha.kunnappilly@mdh.se

† This paper is an extended version of our paper published in Kunnappilly, A.; Marinescu, R.; Seceleanu, C. Assuring Intelligent Ambient Assisted Living Solutions by Statistical Model Checking. In Proceedings of the 2018 International Symposium on Leveraging Applications of Formal Methods (ISoLA 2018), Limassol, Cyprus, 5–9 November 2018.

Received: 30 September 2019; Accepted: 10 November 2019; Published: 19 November 2019

Abstract: Since modern ambient assisted living solutions integrate a multitude of assisted-living functionalities, out of which some are safety critical, it is desirable that these systems are analyzed 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 collection, local and cloud processing schemes, and an intelligent decision support system, which can be easily extended to suit specific architecture categories. Our solution is customizable, therefore, we show three instantiations of the generic model, as simple, intermediate, and complex configurations, 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, which can also account for the probabilistic behavior of such systems, and captures the possibility of component failure. To enable formal analysis, we describe the semantics of the simple and complex architectures within the framework of timed automata. We show that the simple architecture is amenable to exhaustive model checking by employing the UPPAAL tool, whereas for the complex architecture we resort to statistical model checking for scalability reasons. In this case, we apply the statistical extension of UPPAAL, namely UPPAAL SMC. Our work paves the way for the development of formally assured future ambient assisted living solutions.

Keywords: ambient assisted living; Architecture Analysis and Design Language; statistical model checking; UPPAAL SMC

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 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, as in the following cases:

- 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, only an integrated solution would be able to indicate that the potential reason for the fall is in fact the person's low pulse, which in turn may be critical for diagnosis (especially in case of patients having cardiac diseases).

- A high pulse detected during an exercise session: In case of such a scenario, the high pulse is absolutely normal, and hence no alarm should be raised. However, if the activity detection (in this case detecting an 'exercise session') is not combined with pulse monitoring device, a false alarm will be triggered in the scenario.
- 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, due to the inability of the elderly person to answer, the fire alarm may be deemed false and discarded, triggering a potential catastrophe [1,2].

Justified by the above, a timely integration of various assisted living functionalities is veridical. 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 the 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 address 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 modeling the context space and taking the respective actions based on the current context. The system architecture and its DSS are designed as a generic model that can be customized to fit various categories of architectures of different complexities. In this work, we show three 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 [5]. The system architecture, its DSS, and instance models are explained in detail in Section 4.

Our contributions also include a modeling 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 language (AADL) 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 2.1). Once described, the architecture needs to be analyzed formally to check if there are any functional errors and violations of quality-of-service attributes (end-to-end deadlines, fault tolerance, consistency, etc.). To enable this, we transform the architecture specifications into a formal model, in our case, the stochastic timed automata (STA) model, which 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 the art model checker, UPPAAL [6], in the 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 [7]. The analysis results are described in Section 7 and also compared with the results obtained with another formal analysis tool, PRISM [8]. Our approach shows promising results of formally modeling and analyzing complex AAL system specifications, including fault-tolerant and AI-based decisions. A part of this work involving the complex instance of the architecture, its modeling and analysis are presented in the conference paper [5].)

The rest of the paper is organized as follows. In Section 2, we overview the basics of AADL, UPPAAL and UPPAAL SMC. Section 3 describes our proposed methodology. In Section 4, we describe our generic AAL system architecture and its instantiations. We present the AADL modeling constructs and the Agent Annex extension in Section 5. Section 6 describes the formal encoding of the AADL model, and in Section 7, we present the verification results applying the UPPAAL and UPPAAL SMC model checking on AAL system architectures; we also compare the results with those obtained with the PRISM model-checker. Related work is described in Section 9, and conclusions and future work are in Section 10.

2. Preliminaries

In this section, we briefly overview AADL, and the other formal notations and tools used for architecture analysis, that is, timed automata and stochastic timed automata, as well as UPPAAL and UPPAAL SMC.

2.1. The Architecture Analysis and Design Language

AADL [9] 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, Thread Group, etc.), Execution Platform (Device, Bus, Processor, Memory, etc.) and System are used to represent the run-time 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 to 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 functional and error behavior of a component are described by the Behavior Annex (BA) [10] and the Error Annex (EA) [11] 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* [12]. A *component library* creates a repository of component types and implementations with simple hierarchy. It can be established via two packages: (i) the *Interfaces Library* comprising generic components like sensors, actuators, and user-interfaces (UI); and (ii) the *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.

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 [13]; and (2) *statistical model-checking*, using the statistical extension of UPPAAL model checker, UPPAAL SMC [7]. In the following, we overview the semantics of the input models and the mentioned tools.

2.2.1. 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 [14], and is defined by the following tuple:

$$TA = \langle L, l_0, A, V, C, E, I \rangle \tag{1}$$

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, $\Sigma = \Sigma_{\text{i}} \cup \Sigma_{\text{o}}$, 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 time cannot progress in such locations. The latter is 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:

$$STA = \langle TA, \mu, \gamma \rangle. \tag{2}$$

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 invariant, or is unbounded, respectively. With E_l we denote the disjunction of guards g such that $l \xrightarrow{g,o,-} - \in E$ for some output o. Then $d(l, v)$ denotes the infimum delay before the output is enabled, $d(l, v) =$ inf $\{d \in \mathbb{R}_{\geq 0} : v + d \vDash E(l)\}$, whereas $D(l, v)$ =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.

2.2.2. UPPAAL and UPPAAL SMC

The UPPAAL model checker provides exhaustive model-checking of timed-automata models like the ones overviewed in Section 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) [15], and checked by the UPPAAL model checker. UPPAAL supports verification of liveness and safety properties [13]. 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**: $p \leadsto_{\leq t} q$, which means that whenever p holds, q must hold within at most t time units thereafter.

UPPAAL SMC [7], the extension of UPPAAL for statistical model checking, provides the 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) [16] 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".

3. A Framework for Formal Analysis of AAL Systems: Proposed Methodology

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 and 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 4. This architecture design and the requirements in natural language form the input to our analysis framework. As depicted in Figure 1, the framework is composed of the following steps:

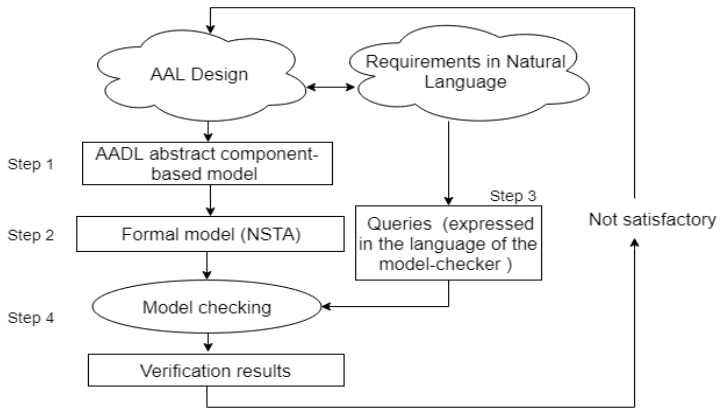
AAL Design
Requirements in Natural Language
Step 1
AADL abstract component-based model
Step 2
Formal model (NSTA)
Step 3
Queries (expressed in the language of the model-checker)
Not satisfactory
Step 4
Model checking
Verification results

Figure 1. Methodology overview. Ambient Assisted Living (AAL), Architecture Analysis and Design language (AADL), stochastic timed automaton (STA), network of interacting STA (NSTA).

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 [17] 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 4. The system modeling in AADL is presented in Section 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 6). We present the semantic anchoring of the generic model and also show 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 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. Consequently, 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 show that requirements are not met, 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.

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

The generic AAL system architecture is presented in Figure 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 the Sensor_A category in Figure 2;
- Non-wearable sensors measuring ambient parameters and health parameters (NW_data), e.g., camera sensors, motion sensors, etc., represented by the Sensor_B category;
- Wearable sensors that detect events (W_event), e.g., fall sensors, marked as the Sensor_C category;
- Non-wearable sensors detecting events (NW_event), e.g., fire sensors, denoted by the Sensor_D category.

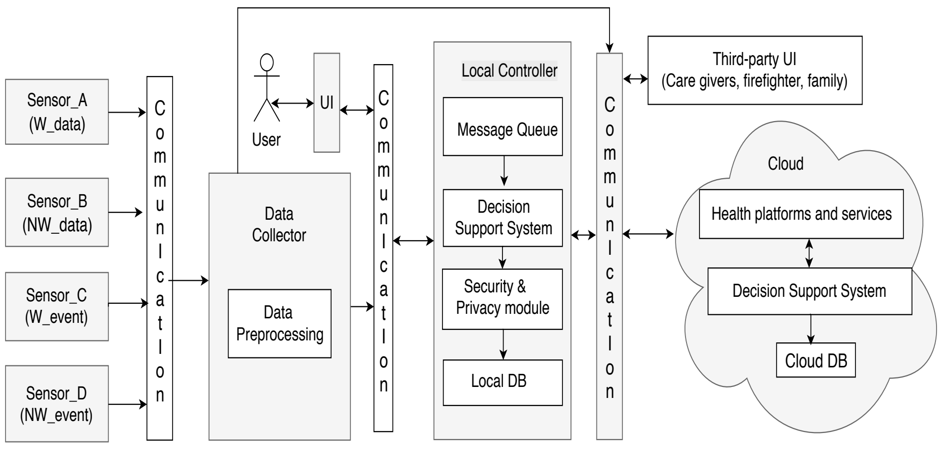

Figure 2. The generic Ambient Assisted Living system architecture.

A particular instantiation of the generic architecture can contain n sensors of each category, respectively, $n \in N$. As depicted in Figure 2, the data from the sensors are collected by the Data Collector unit, which processes the data by assigning labels and 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 Figure 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.

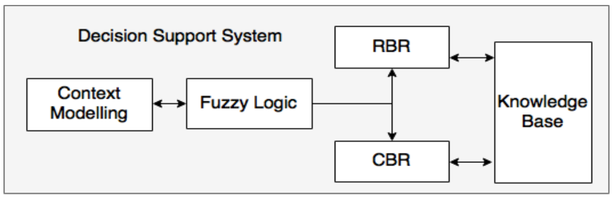

Figure 3. The decision support system (DSS) architecture. Rule-based reasoning (RBR) and case-based reasoning (CBR).

Our DSS architecture is inspired by the work of Zhou et al. [18], 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 to inputs. For instance, CBR reasoning is suited in a clinical decision support system that prescribes medicines/treatment, where the treatment, prescription, and medicine dosage vary for each patient, individually. CBR is an attractive choice due to its reasoning technique resembling more of human problem-solving competence, (i.e., trying to reason about 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, if compared to crisp logic. For instance, the normal pulse range of a person is between 60–100 beats per minute, and a crisp rule-based-reasoning system (Boolean logic) classifies a pulse value of 59.5 or 100.5 beats per minute as an abnormal range (which in reality is not), consequently raising a pulse-deviation alarm to the caregiver. Using fuzzy logic, a degree of membership can be associated to each value, i.e., a pulse value of 59.5 or 100.5 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* [18]. The context-modeling (CM) and the usage of different AI algorithms are depicted in Figure 4.

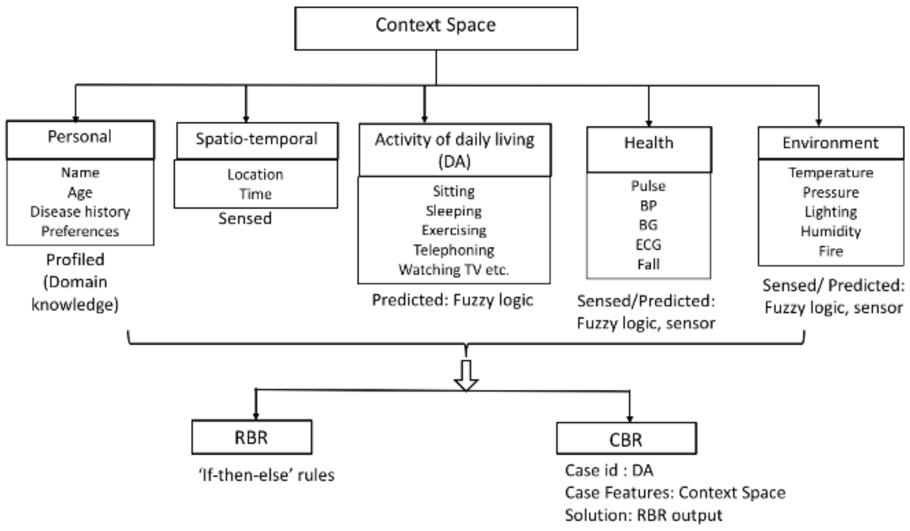

Figure 4. Internals of the DSS architecture (List of Artificial Intelligent techniques).

As indicated, the 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, blood pressure (BP), pulse, blood glucose (BG), 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. *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 [19], and also for determining sensor-data deviations (In order to reduce the complexity of our analysis, we have not explicitly modeled the DA detection using fuzzy logic and have often assumed that the user's DA is known in various scenarios.). To take decisions in various situations, we employ RBR first, and 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 Knowledge Base (KB) stores the context, rules, and cases. The internal structure of the DSS is represented in Figure 4. An example scenario of the DSS reasoning employing different AI techniques is presented in detail in Listing 1 of Section 6.2.

The generic architecture, and its DSS can be instantiated to create a family of AAL architectures that follows similar 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 a 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 Figure 5.
- **Category 2: An intermediate configuration**—This instantiation (see Figure 6) is more complex than the previous one and it contains sensors belonging to all four types 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.

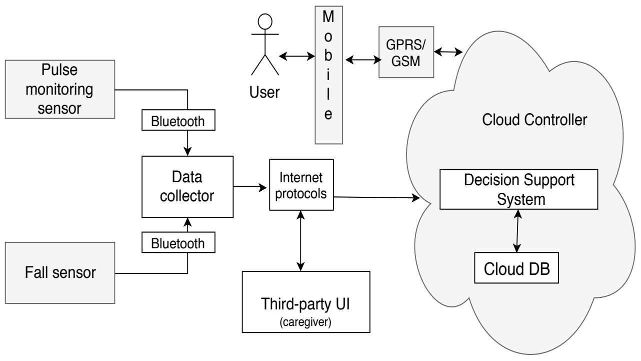

Figure 5. Category 1: A minimal configuration.

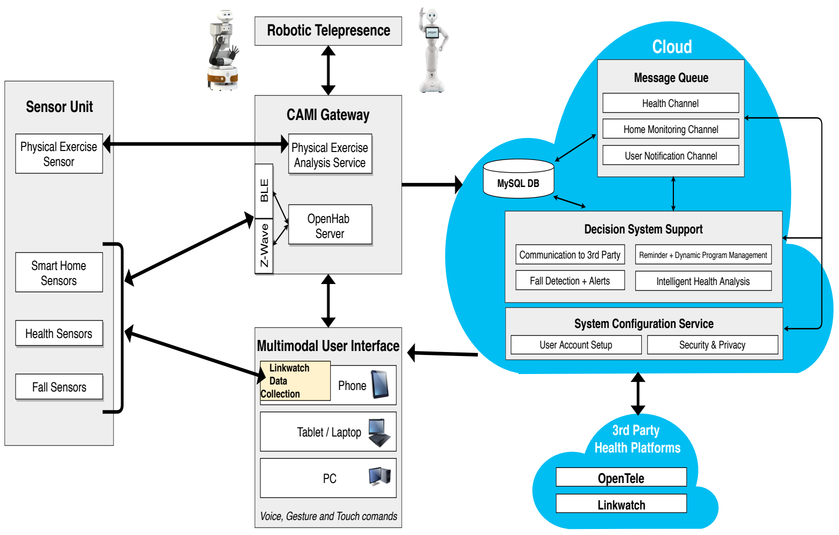

Figure 6. Category 2: An intermediate configuration.

- **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 Figure 7. The latter supports various *sensors* (e.g., a multitude of health and home monitoring sensors like the A&D UA-651 BLE blood pressure sensor [20], Fibaro temperature and motion sensor FGMS-001 [21], Fitbit bracelet [22], Vibby fall detection sensor [23], etc.), data collector, local controller (EXYS9200-SNG [24] referred as *CAMI gateway*), the *CAMI cloud*, and third party health platforms like *Open Tele* [25,26]. There is a set of user interfaces (UI) in CAMI, including robotic platforms (TIAGo [27] and Pepper [28]), 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 [29]. 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.

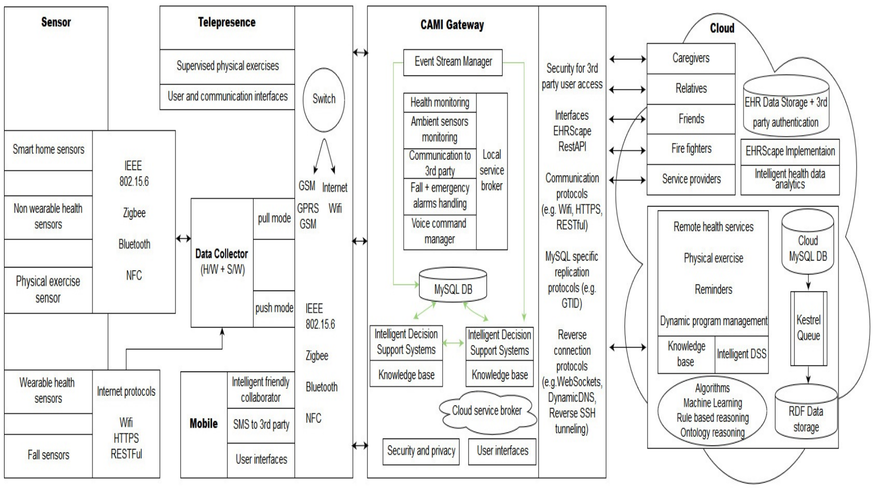

Figure 7. Category 3: A complex configuration: The CAMI AAL System Architecture [2].

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

4.1. Use Case Scenarios and System Requirements

AAL systems should assist elderly users with a variety of health and 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.

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 is 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, this 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 caregiver of the fall event.

However, if Jim needs additional functionality support, then he needs to acquire the CAMI AAL system (Category 3), which can handle additional scenarios to 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:

4.2. Requirements of the Minimal Architecture Model (Category 1):

- **$R1_{Arch1}$**: 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
- **$R2_{Arch1}$**: 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.

4.3. Requirements of the CAMI Architecture (Category 3):

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

- **$R1_{CAMI}$**: 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.
- **$R2_{CAMI}$**: 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.
- **$R3_{CAMI}$**: 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.
- **$R4_{CAMI}$**: 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.
- **$R5_{CAMI}$**: 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.
- **$R6_{CAMI}$**: 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).

5. System Modeling in AADL

The generic architecture, depicted in Figure 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 facilitate the reasoning with functional behavior and errors, we propose a modeling format for both AC and CC as defined below.

5.1. 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 (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 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.). The implementation of 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 a 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$_{\text{CAMI}}$ (Scenario 1), described in Section 2.1. The RBR component type, implementation, BA, and EA are shown in Listing 1. The component type definition specifies its name, category (i.e., "abstract") and interfaces (Lines 2–15). The RBR component type describes that the component 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 17–20) 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 21–28), *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 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 inputs 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 the 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 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 daily activity (DA), case features are the context space and the case solution is the RBR output (refer to Figure 4 to see the behavior of the various AI algorithms). 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 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.

Listing 1: An excerpt from the RBR component in AADL for CAMI.

```
——RBR (Component Type +Implementation)———
abstract RBR
features
input: in event data port;
output: out event data port;
flows
F1 : flow path input -> output;
properties
Dispatch_Protocol => Aperiodic;
property_eventgeneration::AperiodicEventGeneration=>1.0;
property eventgeneration ::Distribution=> Exponential;
property_failure_recovery::FailureRecoveryRate=>1.0;
property_failure_recovery::Distribution=> Exponential;
Compute_Execution_Time =>1s..1s;
end RBR;
abstract implementation RBR.impl
fuzzy_out_pulse:data  fuzzified_data_pulse;
DA: data ADL;
u_profile: data user;
end RBR.impl
——BA——
states
Waiting: initial complete final state;
Operational: state;
transitions
Waiting -[on dispatch input]->Operational
{if (fuzzyo_pulse=high and DA!= exercising and u_prof =cardiac_patient)
{output:= not_caregiver_highpulse}
——EA——
states
Waiting: initial state;
Failed_Transient: state;
Failed_Permanent:state;
LReset: state;
Failed_ep:state;
events
Reset: recover event;
TF: error event;
PF: error event;
Transitions
t1: Waiting -[PF]->Failed_Permanent
t2: Waiting -[TF]->Failed_Transient;
t3: Failed_Transient -[Reset]-> {LReset with 0.9,
Failed_Permanent with 0.1};
t4: LReset-[]->{Waiting with 0.8, Failed_Permanent with 0.2}
properties
EMV2::DurationDistribution => [Duration => 1s..2s; applies to Reset;
EMV2::OccurrenceDistribution =>[ProbabilityValue => 0.9;
Distribution => Fixed;] applies to Reset;
```

5.2. *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 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 2: An excerpt from the DSS component in AADL for CAMI.

```
--DSS Component Type + Implementation--
abstract DSS
features
input: in event data port ;
decision_out: out event data port;
properties
Dispatch_Protocol => Aperiodic;
property_eventgeneration::AperiodicEventGeneration=>10.0;
property eventgeneration ::Distribution=> Exponential;
property_failure_recovery::FailureRecoveryRate=>1.0;
property_failure_recovery::Distribution=> Exponential;
end DSS;
abstract implementation DSS.impl
prototypes
RBR_DSS:abstract RBR;
CBR_DSS:abstract CBR;
CM_DSS:abstract context_model;
subcomponents
RBR:abstract RBR_DSS;
CBR:abstract CBR_DSS;
CM:abstract CM_DSS;
connections
C1: port input -> CM.input;
C2: port CM.output-> RBR.input;
C3: port RBR.output-> CBR.input;
C4: port CBR.output-> decision_out;
--DSS EA--
annex EMV2{**
composite error behavior
[RBR.Failed_Permanent and CBR.Failed_Permanent and
CM.Failed_Permanent] -> Failed_Permanent;
[RBR.Failed_Transient and CBR.Failed_Transient and
CM.Failed_Transient] -> Failed_Transient;
[RBR.Operational or CBR.Operational or
CM.Operational]-> Wait;
EMV2::OccurrenceDistribution =>[ProbabilityValue => 10;
Distribution =>Exponential;] applies to Failed_Permanent,
Failed_Transient, Wait;
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.

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 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 6.2).

6.1. Definition of AADL Components for AAL

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

$$AADL_{\text{Comp}} = \langle Comp_{\text{type}}, Comp_{\text{imp}}, EA, BA \rangle, \tag{3}$$

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* = $IN_p \cup OUT_p$, where IN_p, OUT_p represent the sets of $input\ ports$ and $output\ ports$ respectively, and IN_p, $OUT_p \in$ {*data-ports*, *event-ports*, *event-data-ports*};
- $Flow_{spec} = \langle Flow_{so}, Flow_p, Flow_{si} \rangle$, where $Flow_{so}$, $Flow_p$, $Flow_{si}$ represent flow sources, flow paths and flow sinks respectively. Let $F_{s0} : Flow_{so} \rightarrow OUT_p$ be a function that associates certain OUT_p to $Flow_{so}$ with $Flow_{so} \subseteq OUT_p$, $F_p : Flow_p \rightarrow OUT_p \times IN_p$ be a function that associates and an input and an output to a flow, and $F_{si} : Flow_{si} \rightarrow IN_p$ be a function that associates certain IN_p to $Flow_{si}$, with $Flow_{si} \subseteq IN_p$. For instance, in our AAL architecture, we can define $Flow_{spec}$ for fall events by defining the output port of the fall sensor as $Flow_{so}$, the input port of the cloud DSS as $Flow_{si}$, 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*, $Timing$, $Thread\text{-}related\ properties$, etc. [12]. In this work, we only consider a subset of $Timing$, $Thread\text{-}related\ properties$, and $user\text{-}defined\ properties$, that are represented as follows: $Prop = \{T_p, T_e, Dispatch\ protocol, event\_gen\_dist, failure\_recovery\_dist\}$ where T_p and T_e represent the period and execution-time of the component, respectively, T_p, $T_e \in$ $Timing\ properties$, $Dispatch\ protocol \in \{P, AP\}$ (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; and an aperiodic thread is activated when an event arrives at a port of the thread.), where P represents a Periodic and AP represents an Aperiodic protocol, and $P, AP \in Thread\text{-}related\ properties$, and $event\_gen\_dist, failure\_recovery\_dist \in user\text{-}defined\ properties$ represent the set of user-defined properties used for specifying the occurrence distribution of aperiodic events and failure recovery, respectively.

• $Comp_{imp}$ is defined as $Comp_{imp} = \langle SC, P_t, Con, MSM, Flow_{imp}, ETF \rangle$, where:

- SC represents the set of sub-components of the system with port interfaces (SC_i) and without port interfaces (SC_{Data}), i.e., $SC = SC_{Data} \cup SC_i$;
- P_t denotes the set of *Prototypes* used to define SC via $Fp : P_t \rightarrow SC_i \times SC_{Data}$, a function that associates SC to a P_t, respectively;
- Con represents the set of connections. $F_{con} : Con \rightarrow Features$ is a function that assigns $Features$ to Con;
- MSM is the mode state machine that is modeled by a tuple, as follows: $MSM = \langle M_s, \rightarrow \rangle$, where M_s is the set of states, and $\rightarrow \subseteq M_s \times ev \times M_s$ 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;
- $Flow_{imp}$ are the flow implementations, represented as $Flow_{imp} : SC \rightarrow Flow_{spec}$;
- 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: $EA = \langle E_{flows}, E_{beh}, E_{prop} \rangle$, where:

- E_{flows} denotes the error flows, $E_{flows} = \langle E_{pp}, Err_{so}, Err_p, Err_{si} \rangle$, where E_{pp} describes error propagations, and Err_{so}, Err_p, Err_{si} represents error sources, error paths, and error sinks, respectively; $F_{e1} : Err_{so} \rightarrow OUT_p$ is a function that associates certain output ports with error sources, $F_{e2} : Err_p \rightarrow (IN_p, OUT_p)$ is a function that associates input and output ports via Err_p, $F_{e3} : Err_{si} \rightarrow IN_p$ is a function that assigns certain input ports as error sinks;
- E_{beh} represents error behavior, $E_{beh} = \langle E_s, \rightarrow_e, E_e, EM_{Comp} \rangle$, where E_s represents the set of error states, $\rightarrow_e$ denotes an error transition relation, $\rightarrow_e \subseteq E_s \times E_e \times E_s$, 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'_e be two error states, $s_e, s'_e \in E_s$, and $\rightarrow_e$ 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 the initial state as $s_{0\text{e}} \in E_{\text{s}}$. $F_{E_{\text{pp}}} : Epp \rightarrow (IN_{\text{p}}, OUT_{\text{p}})$ 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 1). In formal semantics, we define it as follows:

$$RBR_{\text{AADL}} = \langle Comp_{\text{type RBR}}, Comp_{\text{imp RBR}}, EA_{\text{RBR}}, BA_{\text{RBR}}, \rangle \tag{4}$$

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-data-ports*},
  - $Flow_{\text{spec RBR}} = \langle Flow_{\text{p}} \rangle$,
  - $Prop_{\text{RBR}} = \{T_{\text{e}}, AP\}$.
- $Comp_{\text{imp RBR}} = \langle SC_{\text{DataRBR}} \rangle$
- $EA_{\text{RBR}} = \{Err_{\text{p}}, E_{\text{s}}, \rightarrow_{\text{e}}, E_{\text{e}}, Dur_{\text{dist}}, Occur_{\text{dist}}\}$
- $BA_{\text{RBR}} = \{B_{\text{s}}, \rightarrow_{\text{b}}\}$.

On the other hand, the DSS in our CAMI architecture is a CC, with multiple subcomponents and hence it is defined by its type, implementation and EA (no BA) as shown in Listing 2. Formally, it can be represented as follows:

$$DSS_{\text{AADL}} = \langle Comp_{\text{type DSS}}, Comp_{\text{imp DSS}}, EA_{\text{DSS}} \rangle \tag{5}$$

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-data-ports*},
  - $Flow_{\text{spec DSS}} = \langle Flow_{\text{p}} \rangle$,
  - $Prop_{\text{DSS}} = \{AP\}$.
- $Comp_{\text{imp DSS}} = \{SC_{\text{DSS}}, Pt_{\text{DSS}}, Con_{\text{DSS}}, Flow_{\text{imp DSS}}\}$, where:
  - $SC_{\text{DSS}} = \{CM, RBR, CBR\}$,

- Pt_{DSS} =$\{CM, RBR, CBR\}$,
- Con_{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.

6.2. Formal Encoding of AADL Components as NSTA

Using the definition of AADL components given in Section 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 2.2), we define a semantic encoding of the AADL components, respectively, in terms of NSTA.

6.2.1. Formal Encoding of AC

Any atomic component in AADL, defined by: $AC = \langle Comp_{\text{typeAC}}, Comp_{\text{implAC}}, EA_{\text{AC}}, BA_{\text{AC}} \rangle$ 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.

• The $\boldsymbol{AC_{iSTA}}$ is defined according to a template STA (see Figure 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:

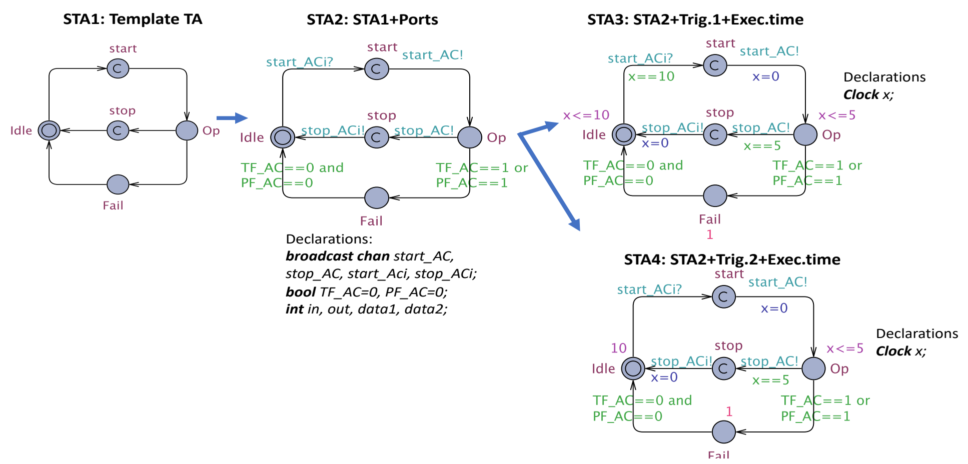

Figure 8. Step-by-step formulation of AC_{iSTA}.

- $V = out\_port \cup in\_port \cup \{PF, TF\} \cup SC_{\text{Data}}$, 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 $SC_{\text{Data}} \in Comp\_imp$;
- $C = \{x\}$ 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;

- $E = \{Idle \xrightarrow{start\_ACi? \wedge x==T_{\mathrm{p}}} start, start \xrightarrow{start\_AC!, x=0} Op,$ $Op \xrightarrow{TF\_AC==1 \vee PF\_AC==1} Fail, Op \xrightarrow{x==T_{\mathrm{e}}, stop\_AC!} stop,$ $stop \xrightarrow{stop\_ACi!, x=0} Idle, Fail \xrightarrow{TF\_AC==0 \wedge PF\_AC==0} Idle,$ $Fail \xrightarrow{TF\_AC==1 \wedge PF\_AC==1} Fail\}$, where E is defined by the template populated with A and guards that ensure the correctness of transitions.
- $I(Idle)$=$(x \leq T_{\mathrm{p}})$, if the dispatch protocol associated with AC is periodic, and $I(Op) = x \leq T_{\mathrm{e}}$, where T_{p} and T_{e} represent the period and execution-time of AC;
- $P(Idle) = \mu_1$, 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;
  - $A = \{start\_AC?, stop\_AC?\} \cup \{action_{\mathrm{BA,EA}}(), TF = 0, TF\_AC = 1, PF\_AC = 1, reset\_AC = 0, reset\_AC = 1, err\_pAC = 0, err\_pAC = 1, err\_p = 1, y = 0\}$, where A is composed of the actions defined in BA and EA ($\mathrm{action}_{\mathrm{BA,EA()}}$), 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;
  - $E = \{Wait \xrightarrow{start\_AC?} L1, L1 \xrightarrow{TF\_AC=1, err\_pAC=1} TrF, L1 \xrightarrow{PF\_AC=1, err\_pAC=1} PrF, L1 \rightarrow L2, L2 \rightarrow Op, Op \xrightarrow{stop\_AC?, action_{\mathrm{BA}}()} Wait, TrF \xrightarrow{reset\_AC=1, y=0} LReset, TrF \xrightarrow{PF\_AC=1, err\_pAC=1, reset\_AC=0} PrF, LReset \xrightarrow{TF\_RBR=0, err\_pAC=0, reset\_AC=0} Wait, LReset \xrightarrow{PF\_AC=1, err\_pAC=1, reset\_AC=0} PrF, Wait \xrightarrow{err\_p==1} Fail\_ep\}$, where E consists of the transitions in EA, BA and those between $L1$ and $L2$;
  - $I(LReset)$= $(y \leq Dur_{\mathrm{dist(Reset)}})$;
  - $P(Wait) = \mu$, 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. □

6.2.2. Formal Encoding of CC

The formal encoding of a CC defined by the tuple: $CC = \langle Comp_{\mathrm{typeCC}}, Comp_{\mathrm{implCC}}, EA_{\mathrm{CC}} \rangle$ is also a network of two synchronized STA, $CC_{\mathrm{NSTA}} = CC_{\mathrm{iSTA}} || CC_{\mathrm{aSTA}}$, 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 $(Comp_{\mathrm{typeCC}}, Comp_{\mathrm{implCC}})$, as follows:
  - $L = \{Wait, Fail\} \bigcup_{i=1}^{n} \{L_{\mathrm{i}}Sync\} \bigcup_{i=1}^{n} \{SC_{\mathrm{i}}\}$, 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 two edges, $l \longrightarrow L_iSync$ and $L_iSync \longrightarrow l'$, where l, l′$\in L$ are locations created based on the sub-components for which the connections are defined, and $L_iSync \in L$ is a location created for synchronization;
- $V = out\_port \cup in\_port \cup \{PF, TF\} \cup SC_{\text{Data}}$, 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 $SC_{\text{Data}} \in Comp\_imp$;
- $C = \{x\}$ if $T_{\text{p}} \neq \emptyset$;
- A is defined based on the updates defined by MSM, the updates defined by $Flow_{\text{imp}}$, 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;
- $I(Wait)$=$(x \leq T_{\text{p}})$ if $T_{\text{p}} \neq \emptyset$;
- $P(l) = \mu$, where $l \in L$ and μ is defined by $Prop$.

• $\boldsymbol{CC_{aSTA}}$ is defined as follows:

- $L = E_{\text{s}} \in EA$, $l_0 = s_{0\text{e}} \in E_{\text{s}}$, where E_{s} is the set of states of EA;
- $E = \longrightarrow_{\text{e}}$;
- $A = \{TF\_CC = 1, TF\_CC = 0, PF\_CC = 1\}$;
- 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 $Occur_{\text{dist}} \in E_{\text{prop}}$.

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 Equation (4), is mapped into an NSTA (RBR$_{\text{NSTA}}$) as follows: RBR_{NSTA}=$RBR_{\text{iSTA}}||RBR_{\text{aSTA}}$ (Figure 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.

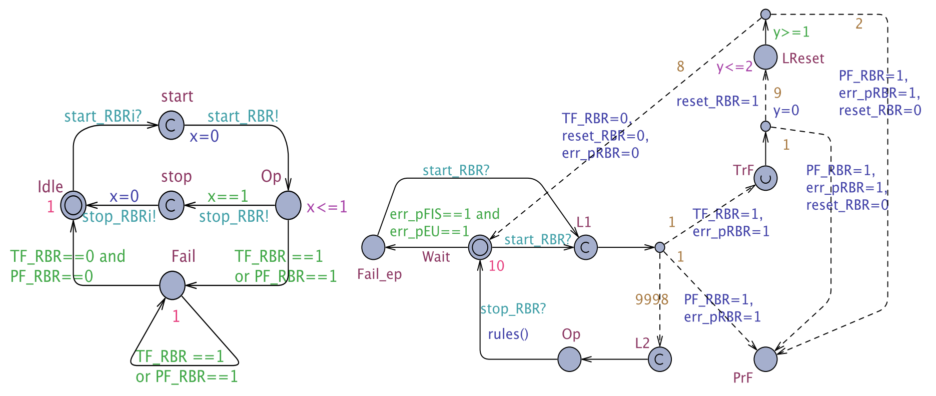

Figure 9. The STA for the RBR. (**a**) Interface STA (RBR_{iSTA}); (**b**) Annex STA (RBR_{aSTA}).

• The RBR_{iSTA} is formally represented as a tuple, where:

- $L = \{Idle, Start, Op, Fail\}, l0 = \{Idle\}$

- $A = \{start\_RBRi?, start\_RBR!, stop\_RBR?\} \cup \{x = 1\}$
- $V = \{out\_port, in\_port, PF\_RBR, TF\_RBR\}$
- $C = \{x\}$
- $E = \{Idle \xrightarrow{start\_RBRi?} start, start \xrightarrow{start\_RBR!, x=0} Op,$ $Op \xrightarrow{TF\_RBR==1 \vee PF\_RBR==1} Fail, Op \xrightarrow{x==1, stop\_RBR!} Idle, Fail$ $\xrightarrow{TF\_RBR==0 \wedge PF\_RBR==0} Idle, Fail \xrightarrow{TF\_RBR==1 \wedge PF\_RBR==1} Fail\}$
- $I(Op)$=$(x \leq 1)$
- $P(Idle) = 1$, $P(Fail) = 1$, given by γ

• RBR_{aSTA} is defined in a similar way:

- $L = \{Wait, Op, TrF, PrF, Fail\_ep, LReset, L1, L2, LSync\}, \{l0 = Wait\}$
- $A = \{start\_RBR?, stop\_RBR?, stop\_RBRi!\} \cup \{rules(), TF\_RBR{=}\{0, 1\},$ $PF\_RBR{=}\{1\}, reset\_RBR{=}\{0, 1, \}, err\_pRBR{=}\{0, 1\}, err\_p{=}\{1\}, y{=}0\}$
- $V = \{PF\_RBR, TF\_RBR, reset\_RBR, err\_pRBR, err_p\}$
- $C = \{y\}$
- $E = \{Wait \xrightarrow{start\_RBR?} L1, L1 \xrightarrow{TF\_RBR=1, err\_pRBR=1} TrF, L1$ $\xrightarrow{PF\_RBR=1, err\_pRBR=1} PrF, L1 \rightarrow L2, L2 \rightarrow Op, Op \xrightarrow{stop\_RBR?, rules()} Lsync, Lsync \xrightarrow{stop\_RBRi!}$ $Wait, TrF \xrightarrow{reset\_RBR=1, y=0} LReset,$ $TrF \xrightarrow{PF\_RBR=1, err\_pRBR=1, reset\_RBR=0} PrF,$ $LReset \xrightarrow{TF\_RBR=0, err\_pRBR=0, reset\_RBR=0} Wait,$ $LReset \xrightarrow{PF\_RBR=1, err\_pRBR=1, reset\_RBR=0} PrF, Wait \xrightarrow{err\_p==1} Fail\_ep\}$
- $I(LReset) = y \leq 2$
- $P(Wait) = 10$, given by μ
- $L1 \xrightarrow{0.9998} L2$, $L1 \xrightarrow{0.001} TrF$, $L1 \xrightarrow{0.001} PrF$, assigned by γ

Similarly, the DSS_{AADL}, shown in Listing 2, and represented by Equation (5), is mapped into an NSTA: $DSS_{\mathrm{AADL}} \rightsquigarrow DSS_{\mathrm{NSTA}}$=$DSS_{\mathrm{iSTA}}||DSS_{\mathrm{aSTA}}$ (Figure 10), where DSS_{iSTA} is the so-called "Interface STA" of DSS, which corresponds to $Comp_{\mathrm{type\ DSS}}$ and $Comp_{\mathrm{impl\ DSS}}$, whereas DSS_{aSTA} is the "Annex STA" that encodes the EA of CC.

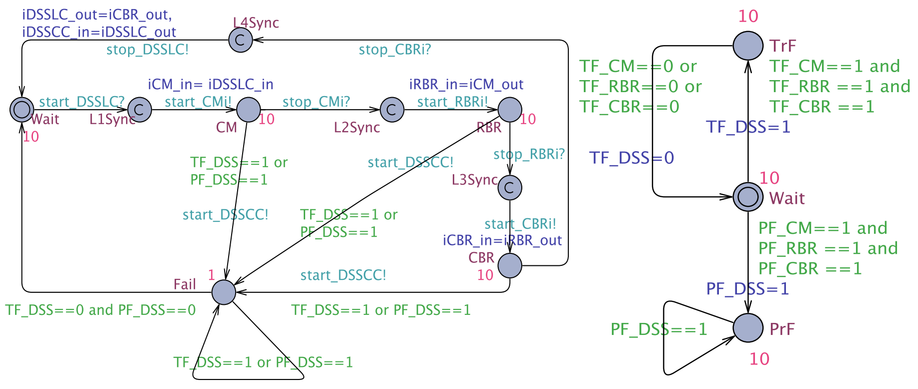

Figure 10. The STA for the DSS. (**a**) Interface STA (DSS_{iSTA}); (**b**) Annex STA (DSS_{aSTA}).

• 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

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

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

It should be noted that in 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 (four sensors: one for detecting pulse data deviation, two for fall detection and one for fire detection, data collector, Message Queue, RBR, CBR, daily activity detection, fuzzy logic) and the remaining 12 by encoding six CC (Local Processor, Cloud Processor, DSS (Local and Cloud), and 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 is comprised of only 18 STAs and is shown to be scalable with exhaustive analysis.

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 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.

7.1. Exhaustive Verification of the Minimum Configuration Using UPPAAL

The results of the exhaustive verification of the minimum configuration architecture using the UPPAAL model checker are tabulated in Table 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 Figure 11. As described, we start the monitoring clock *s1* when the pulse sensor produces the data, marked by the 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 $R2_{\text{Arch1}}$.

Table 1. UPPAAL analysis results for the minimum configuration architecture.

REquation	**Query**	**Result**
$R1_{\text{Arch1}}$	$(110 \leq sd\_w.data\_val \leq 300$ *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
$R2_{\text{Arch1}}$	$(se\_w.fall == 1$ *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

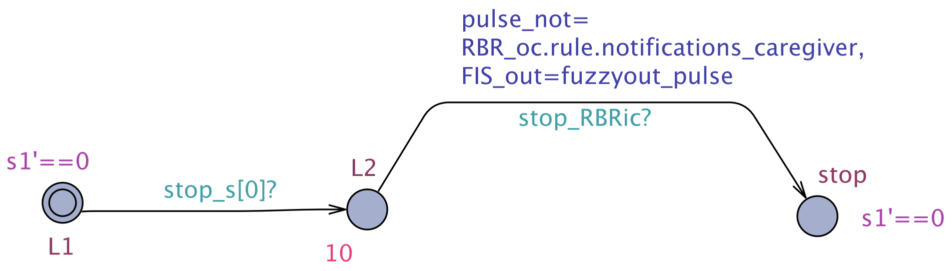

Figure 11. The monitor automaton for requirement $R1_{\text{Arch1}}$.

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 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. However, its equally important to mention that if the real-time constraints are relaxed, for instance, if we check if these requirements are met within 10 s, the model-checker obviously returns a fail for the corresponding query (shown in Table 2).

Table 2. UPPAAL analysis: A case where real-time constraints are not met.

REquation	**Query**	**Result**
$R1_{\text{Arch1}}$	$(110 \leq sd\_w.data\_val \leq 300$ *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*$M\_pulse.s1 \leq 10$	Fail

7.2. Statistical Verification of the CAMI architecture Using UPPAAL SMC

In the 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 [7], 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 Figure 12. As shown, we start the monitoring clock $s1$ when the fire 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 local DSS or the cloud DSS. Similar monitors are employed for $R2_{\text{CAMI}}$, $R3_{\text{CAMI}}$, $R4_{\text{CAMI}}$, and$R5_{\text{CAMI}}$.

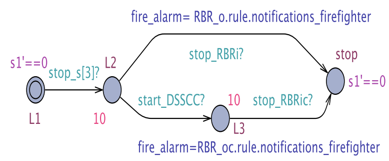
fire_alarm= RBR_o.rule.notifications_firefighter
s1'==0
stop_s[3]?
L2
stop_RBRi?
stop
s1'==0
L1
10
start_DSSCC?
10
stop_RBRic?
L3
fire_alarm=RBR_oc.rule.notifications_firefighter

Figure 12. The monitor automaton for requirement $R1_{\text{CAMI}}$.

The verification results are tabulated in Table 3. The CAMI architecture model satisfies all the requirements with probabilities close to 1 with a high confidence within four 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 by the fact 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.

Table 3. UPPAAL SMC analysis results of CAMI.

REquation	Query	Result	Runs
$R1_{\text{CAMI}}$	$Pr[<=1000]([]((M\_fire.fire\_alarm == 1)$ $imply\ (se\_nw.fire == 1$ 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
$R2_{\text{CAMI}}$	$Pr[<=1000]([]((M\_fall.fall\_not == 7)$ $imply\ ((se\_w.fall == 1$ or $sd\_nw.data\_val == 1)$ $and(M\_fall.s1 <= 20))))$	Pr [0.99975,1] confidence 0.998	3868
	$Pr[<=1000](<>\ (M\_fire.fire\_alarm == 1))$	Pr [0.99975,1] confidence 0.998	4901
$R3_{\text{CAMI}}$	$Pr[<=1000]([](\ M\_firefall.fire\_not == 2\ and$ $M\_firefall.fall\_not == 2\ imply$ $((se\_w.fall == 1$ 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
$R4_{\text{CAMI}}$	$Pr[<=1000]([]((M\_pulse.pulse\_not == 3)$ $imply\ (110 <= sd\_w.data\_val <= 300\ and$ $M\_pulse.FIS\_out == 3\ and\ ADL == 1\ and$ $upro.disease\_history == 3$ 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

Table 3. *Cont.*

REquation	Query	Result	Runs
R5$_{\text{CAMI}}$	$Pr[<=1000]([](M\_consistency.stop\ imply\ (RBR\_om == iCBRCC_m)))$	Pr [0.99975,1] confidence 0.998	3868
	$Pr[<=1000](<>(M\_consistency.stop))$	Pr [0.99975,1] confidence 0.998	5777
R6$_{\text{CAMI}}$	$Pr[<=1000]([](INT\_CC.DSSCC\ imply\ PF\_DSS == 1))$	Pr [0.99975,1] confidence 0.998	3868
	$Pr[<=1000](<>(INT\_CC.DSSCC))$	Pr [0.01,0.04] confidence 0.998	2885

7.3. Comparison of the Proposed Approach With the Model-Checker PRISM

In this section, we show a small-scale modeling of the CAMI architecture using the model checker, PRISM [8] to compare the results that we have obtained with UPPAAL SMC. For details regarding the transformation of AADL to PRISM, please refer our previous work [30]. To ensure scalability of the analysis, we model only the scenarios of fire and fall where the rule-based reasoning takes the action of forwarding the respective alerts to firefighter and caregivers, respectively (R1$_{\text{CAMI}}$, R2$_{\text{CAMI}}$ and R3$_{\text{CAMI}}$).

In Listing 3, we show an excerpt of the RBR module modeled as a Probabilistic Timed Automaton (PTA) in PRISM model checker. In line 3, we define the variable s representing the state of the system: s = 0 (*Idle*), s = 1 (*Op*), s = 3 (*TrF*), s = 4 (*PrF*), and s = 5 (*Fail_ep*). In the following lines, we define the event variables for transient failure (TF_RBR), permanent failure (PF_RBR), failure due to error propagation (Fail_ep_RBR) and the reset event (reset_RBR). We also define a clock variable x to model the RBR's execution time (an invariant associated with state *Op*). In lines 13–22, we model the functional and error behavior of the component as discussed in the previous sections. We show the cases of a fire event and fall event generated separately, where subsequent alerts to firefighter and caregiver informing of the respective events are sent. We also illustrate the case where fire and fall events occur together, where both the events need to be sent to both caregiver and firefighter.

Listing 3: An excerpt of the PRISM model of an RBR

```
1  pta
2  module RBR
3  s: [0..4] init 0;
4  PF_RBR:[0..1] init 0;
5  TF_RBR:[0..1] init 0;
6  Fail_ep_RBR:[0..1] init 0;
7  reset_RBR: [0..1] init 0;
8  //states 0 -Idle, 1-Op, 2-Fail_Transient, 3-Fail_Permanent, 4-Fail_ep
9  x: clock;
10 invariant
11 (s=1 => x<=1)
12 endinvariant
13 [1]s =0 & fire_s=1 &fall_s=0 & Fail_ep_RBR=0-> 0.998:(s'=1) & (x'=0) +
14 0.001:(s'=2) &(x'=0) +=.001(s'=3) &(x'=0);
15 s=2 ->TF_RBR'=1; s=3 -> PF_RBR'=1;
16 [2]s=1  & x=1 & fire_s=1 &fall_s=0 & Fail_ep_RBR=0 -> (s'=0) & (fire_alert=1) & (x'=0);
17 [3]s =0 & fire_s=0 &fall_s=1 & Fail_ep_RBR=0> 0.998:(s'=1) & (x'=0) +
18 0.001:(s'=2) &(x'=0) +0.001(s'=3) &(x'=0);
19 [4]s=1  & x=1 & fire_s=0 &fall_s=1 & Fail_ep_RBR=0 -> (s'=0) & (fall_alert=1) & (x'=0);
20 [5]s =0 & fire_s=1 &fall_s=1-> 0.998:(s'=1) & (x'=0) +
21 0.001:(s'=2) &(x'=0) +=.001(s'=3) &(x'=0);
22 [6]s=1  & x=1 & fire_s=1 &fall_s=1 & Fail_ep_RBR=0-> (s'=0) & (fall_alert=2) & (fire_alert=2) (x'=0);
23 endmodule
```

The analysis results are presented in Table 4. The CAMI requirements are formulated as Probabilistic Computation Tree Logic (PCTL) queries. Moreover, since the PRISM model checker returns the result for the initial state of the model by default, we employ *filters* to verify the properties over all states. R1$_{\text{CAMI}}$ ensures that if a fall event is raised by fall sensor, then the fall alert is communicated to the caregiver within 20 time units, provided that none of the components has failed. Similarly, in R2$_{\text{CAMI}}$, we show the case of a fire event being communicated to a firefighter within 20 time units. In R3$_{\text{CAMI}}$, we show the case of fire and fall events simultaneously raised, where we need to communicate both events to firefighter and caregiver within a real-time deadline of 20 time units.

Table 4. PRISM verification results.

REquation	Query	Result	Time (s)
R1$_{\text{CAMI}}$	$filter(forall, fall\_s = 1\&fire\_s = 0\&PF\_RBR = 0 \rightarrow P \geq 0.999$ $[F((fall\_alert = 1)\&(x \leq 20)\&(fall\_fail = 0)\&(DC\_fail = 0)]$	satisfied	2001.7
R2$_{\text{CAMI}}$	$filter(forall, fall\_s = 0\&fire\_s = 1\&PF\_RBR = 0 \rightarrow P \geq 0.999$ $[F((fire\_alert = 1)\&(x \leq 20)\&(fire\_fail = 0)\&(DC\_fail = 0)]$	satisfied	2001.7
R3$_{\text{CAMI}}$	$filter(forall, fall\_s = 1\&fire\_s = 1\&PF\_RBR = 0 \rightarrow P \geq 0.999$ $[F((fire\_fall\_alert = 2)\&(x \leq 20)\&(fire\_fail = 0)\&(fall\_fail = 0)$ $\&(DC\_fail = 0)]$	satisfied	3500.13

We can start the comparison of UPPAAL, PRISM, and UPPAAL SMC verification with the following known facts: (a) PRISM allows exhaustive model-checking of probabilistic systems (b) For UPPAAL, although the results are exhaustive, the probabilistic view of the system is discarded and (c) UPPAAL SMC allows us to analyze probabilistic systems, however the analysis is simulation-based, that is, the guarantees are obtained by simulating the system for a finite number of runs and hence not exhaustive or fully guaranteed. In our case, the major disadvantage we found with PRISM was that it did not scale well and hence we had to limit the complexity of our CAMI architecture for analysis purposes. Moreover, it lacks a graphical GUI, as compared to UPPAAL systems and do not provide simulation-based analysis for PTA models, thereby limiting the diagnosis of the models.

8. 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 [31] for automated modeling, and provides some preliminary architecture-level analysis. It also allows us to model the behavior of the architecture components via a behavior annex and encode the probabilities of failure of various components, via the error annex. However, AADL 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) involving exhaustive model checking, with the UPPAAL tool (2) involving statistical model checking with UPPAAL SMC, and (3) involving probabilistic model checking with PRISM. The analysis approaches are chosen based on the system complexity. If the architecture model is scalable with exhaustive model checking, then the latter can be applied. Although the exhaustive verification results obtained by UPPAAL are accurate, one cannot take into account the probabilistic behavior of our systems. In comparison, PRISM handles probabilistic systems and carries out an exhaustive analysis, however its scalability is considerably reduced if compared to UPPAAL SMC. In the case of complex models that need to be analyzed for stochastic behaviors, the user can opt for simulation-based approaches, although it does not yield a 100% accuracy. The verification results shown in this paper are specific to our architecture models, however one can use the approach to verify any set of requirements for various architecture types created based on the generic architectural model defined in this work. In case of exhaustive model-checking, the results are derived assuming that all components are operational such that we devoid the system of its probabilistic failure behavior. For statistical 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. Nevertheless, taking into account the wide variety of available sensors and other components, we can easily adapt the values to account for the 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 can be extended 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 integrated AAL solutions.

9. 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 [32]. Moreover, many functionalities that need to be tackled by AAL solutions are of a safety-critical nature, e.g., health emergencies like cardiac arrest, falls of the elderly, and home emergencies like fires at home, etc. [33], 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 Systems (MAS) [34–36], and Cloud-based [37,38] and Internet-of-Things (IoT) centric solutions [39].

- **Agent-based architectures:** These are the most commonly used architectures for AAL applications, based of their flexibility, autonomy, adaptability, better response, and service continuity due to their distributed nature. Some examples of health-care frameworks that rely on a distributed agent architecture are described in the literature [34,40]. However, the agent-based architectures have some drawbacks: (i) restricted communication protocols for agent communication and the delay overhead in taking a collective decision; and (ii) maintaining the consistency.
- **Cloud-based AAL solutions:** This category includes AAL solutions that leverage the potential of cloud computing for context modeling, intelligent decision making, and data-storage usage. Our architecture follows the design paradigms of Cloud-based AAL solutions, where the cloud is utilized for intelligent, context-aware decision making, also as a data store, and it is also augmented with local processing schemes to guarantee real-time properties. In many situations, cloud services cannot guarantee hard-real time properties, the cloud being a backup that gets activated only when the primary one 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 [41]. In another interesting work, Rodrigues et al. [4] performed a dependability analysis of AAL architectures using UML and PRISM. Other interesting research work uses temporal reasoning [3,42] and Markov Decision Processes to formally verify the reliability of AAL systems [43]. 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 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 such languages are commonly employed in the design of 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 [44–46]. Although these approaches rely on 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. [47], where the authors have used the 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. used the SLIM Language, which is strongly based on AADL and is specific to avionics and the 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, which enables us to represent the functional and error behaviors, together with architecture model. The abstract component-based modeling also brings extensiblity 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, whereas we also evaluate properties like reliability, data consistency, etc., along with timeliness. Another interesting work [48], possibly carried out in parallel with our work, employed 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 with respect to specifying complex cyber physical systems considering the uncertainties of the physical environment. Unlike our model, the authors used 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 each other, yet our approach is 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. 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 networks of stochastic timed automata.

We show that the resulting formal models are analyzable exhaustively with UPPAAL/PRISM 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 ones, 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 with respect to 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 formal modeling and analysis of distributed versions of the integrated architectures for AAL.

Author Contributions: A.K. was the main driver of the paper. The other two authors provided ideas Writing—review and editing the modeling and verification of the AAL system, as well as feedback for the paper.

Funding: This research was funded by joint EU/Vinnova project grant CAMI, AAL-2014-1-087.

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

References

1. Kunnappilly, A.; Seceleanu, C.; Lindén, M. Do We Need an Integrated Framework for Ambient Assisted Living? In Proceedigns of the Ubiquitous Computing and Ambient Intelligence: 10th International Conference, UCAmI 2016, San Bartolomé de Tirajana, Gran Canaria, Spain, 29 November–2 December 2016; Springer: Cham, Switzerland 2016; Part II 10, pp. 52–63.
2. Kunnappilly, A.; Sorici, A.; Awada, I.A.; Mocanu, I.; Seceleanu, C.; Florea, A.M. A Novel Integrated Architecture for Ambient Assisted Living Systems. In Proceedigns of the 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), Turin, Italy, 4–8 July 2017; Volume 1, pp. 465–472.
3. Augusto, J.C.; Nugent, C.D. The use of temporal reasoning and management of complex events in smart homes. In Proceedings of the 16th European Conference on Artificial Intelligence, Valencia, Spain, 22–27 August 2004; IOS Press: Amsterdam, The Netherlands, 2004; pp. 778–782.
4. Rodrigues, G.N.; Alves, V.; Silveira, R.; Laranjeira, L.A. Dependability analysis in the ambient assisted living domain: An exploratory case study. *J. Syst. Softw.* **2012**, *85*, 112–131. [CrossRef]
5. Kunnappilly, A.; Marinescu, R.; Seceleanu, C. Assuring intelligent ambient assisted living solutions by statistical model checking. In *International Symposium on Leveraging Applications of Formal Methods*; Springer: Cham, Switzerland, 2018; pp. 457–476.
6. Bengtsson, J.; Larsen, K.; Larsson, F.; Pettersson, P.; Yi, W. UPPAAL—A tool suite for automatic verification of real-time systems. In *International Hybrid Systems Workshop*; Springer: Berlin/Heidelberg, Germany, 1995; pp. 232–243.
7. David, A.; Larsen, K.G.; Legay, A.; Mikučionis, M.; Poulsen, D.B. Uppaal SMC tutorial. *Int. J. Softw. Tools Technol. Transf.* **2015**, *17*, 397–415. [CrossRef]
8. Kwiatkowska, M.; Norman, G.; Parker, D. PRISM: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, Springer: Berlin/Heidelberg, Germany, 2002; pp. 200–204.
9. Feiler, P.H.; Lewis, B.; Vestal, S.; Colbert, E. 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*; Springer: Boston, MA, USA, 2005; pp. 3–15.
10. Frana, R.; Bodeveix, J.P.; Filali, M.; Rolland, J.F. The AADL behaviour annex–experiments and roadmap. In Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems, Auckland, New Zealand, 11–14 July, 2007; pp. 377–382.
11. Delange, J.; Feiler, P. Architecture fault modeling with the AADL error-model annex. In Proceedings of the 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Verona, Italy, 27–29 August 2014; pp. 361–368.
12. Feiler, P.H.; Gluch, D.P. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*; Addison-Wesley: Boston, MA, USA, 2012.
13. Larsen, K.G.; Pettersson, P.; Yi, W. UPPAAL in a nutshell. *Int. J. Softw. Tools Technol. Transf.* **1997**, *1*, 134–152. [CrossRef]
14. Alur, R.; Courcoubetis, C.; Dill, D. Model-checking in dense real-time. *Inf. Comput.* **1993**, *104*, 2–34. [CrossRef]
15. Alur, R.; Courcoubetis, C.; Dill, D. Model-checking for real-time systems. In Proceedings of the 1990 Fifth Annual IEEE Symposium on Logic in Computer Science, Philadelphia, PA, USA, 4–7 June 1990; pp. 414–425.
16. Bulychev, P.E.; David, A.; Larsen, K.G.; Legay, A.; Li, G.; Poulsen, D.B. Rewrite-Based Statistical Model Checking of WMTL. *RV* **2012**, *7687*, 260–275.

17. The Architecture Analysis & Design Language (AADL): An Introduction. Available online: https://people.cs.clemson.edu/~johnmc/courses/cpsc875/resources/06tn011.pdf (accessed on 19 November 2019).
18. Zhou, F.; Jiao, J.R.; Chen, S.; Zhang, D. A case-driven ambient intelligence system for elderly in-home assistance applications. *IEEE Trans. Syst. Man Cybern. Part C (Appl. Rev.)* **2011**, *41*, 179–189. [CrossRef]
19. Medjahed, H.; Istrate, D.; Boudy, J.; Dorizzi, B. Human activities of daily living recognition using fuzzy logic for elderly home monitoring. In Proceedings of the 2009 IEEE International Conference on Fuzzy Systems, Jeju Island, Korea, 20–24 August 2009; pp. 2001–2006.
20. UA651 BP Sensor. Available online: http://www.andmedical.com.au/products-service/value-ua-651 (accessed on 16 March 2019).
21. Fibaro motion sensor. Available online: https://manuals.fibaro.com/content/manuals/en/FGMS-001/FGMS-001-EN-T-v2.0.pdf (accessed on 16 March 2019).
22. Fitbit. Available online: https://www.fitbit.com/se/home (accessed on 16 March 2019).
23. Vibby Fall Detection Sensors. Available online: http://www.vitalbase.co.uk (accessed on 16 March 2019).
24. CAMI Gateway. Available online: https://eclexys.com/wp-content/uploads/2019/01/Exys9200-SNG-Brochure.pdf (accessed on 16 March 2019).
25. OpenTele. Available online: https://www.opentelehealth.com (accessed on 15 January 2018).
26. Linkwatch. Available online: https://www.linkwatch.se (accessed on 15 January 2018).
27. TIAGo Robotic Platform. Available online: http://tiago.pal-robotics.com (accessed on 16 March 2019).
28. Pepper Robot. Available online: https://www.softbankrobotics.com/emea/en/pepper (accessed on 16 March 2019).
29. Rabbit MQ Message Broker. Available online: https://www.rabbitmq.com (accessed on 16 March 2019).
30. Kunnappilly, A.; Cai, S.; Marinescu, R.; Seceleanu, C. Architecture Modelling and Formal Analysis of Intelligent Multi-Agent Systems. In Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering, Heraklion, Crete, Greece, 4–5 May 2019.
31. OSATE—Open Source AADL Test Environment. Available online: http://osate.github.io/ (accessed on 15 May 2018).
32. Li, R.; Lu, B.; McDonald-Maier, K.D. Cognitive assisted living ambient system: A survey. *Digit. Commun. Netw.* **2015**, *1*, 229–252. [CrossRef]
33. Rashidi, P.; Mihailidis, A. A survey on ambient-assisted living tools for older adults. *IEEE J. Biomed. Health Inform.* **2013**, *17*, 579–590. [CrossRef] [PubMed]
34. De Paz, J.; Rodríguez, S.; Bajo, J.; Corchado, J.; Corchado, E. OVACARE: A multi-agent system for assistance and health care. In *Knowledge-Based and Intelligent Information and Engineering Systems*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 318–327.
35. Isern, D.; Sánchez, D.; Moreno, A. Agents applied in health care: A review. *Int. J. Med Inform.* **2010**, *79*, 145–166. [CrossRef] [PubMed]
36. Nealon, J.; Moreno, A. Agent-based applications in health care. In *Applications of Software Agent Technology in the Health Care Domain*; Birkhäuser: Basel, Switzerland, 2003; pp. 3–18.
37. Ahmed, M.U.; Björkman, M.; Lindén, M. A generic system-level framework for self-serve health monitoring system through internet of things (iot). *Stud. Health Technol. Inform.* **2015**, *211*, 305–307. [PubMed]
38. Forkan, A.; Khalil, I.; Tari, Z. CoCaMAAL: A cloud-oriented context-aware middleware in ambient assisted living. *Future Gener. Comput. Syst.* **2014**, *35*, 114–127. [CrossRef]
39. Dohr, A.; Modre-Osprian, R.; Drobics, M.; Hayn, D.; Schreier, G. The Internet of Things for Ambient Assisted Living. *ITNG* **2010**, *10*, 804–809.
40. Tapia, D.I.; Rodrıguez, S.; Corchado, J.M. A distributed ambient intelligence based multi-agent system for Alzheimer health care. In *Pervasive Computing*; Springer, London, UK, 2009; pp. 181–199.
41. Parente, G.; Nugent, C.D.; Hong, X.; Donnelly, M.P.; Chen, L.; Vicario, E. Formal modeling techniques for ambient assisted living. *Ageing Int.* **2011**, *36*, 192–216. [CrossRef]
42. Magherini, T.; Fantechi, A.; Nugent, C.D.; Vicario, E. Using temporal logic and model checking in automated recognition of human activities for ambient-assisted living. *IEEE Trans. Hum.-Mach. Syst.* **2013**, *43*, 509–521. [CrossRef]
43. Liu, Y.; Gui, L.; Liu, Y. MDP-based reliability analysis of an ambient assisted living system. In *International Symposium on Formal Methods*; Springer: Cham, Switzerland, 2014; pp. 688–702.

44. Besnard, L.; Gautier, T.; Le Guernic, P.; Guy, C.; Talpin, J.P.; Larson, B.; Borde, E. Formal semantics of behavior specifications in the architecture analysis and design language standard. In *Cyber-Physical System Design from an Architecture Analysis Viewpoint*; Springer: Cham, Switzerland, 2017; pp. 53–79.
45. Hamdane, M.E.; Chaoui, A.; Strecker, M. From AADL to timed automaton-A verification approach. *Int. J. Softw. Eng. Appl.* **2013**, *7*.
46. Johnsen, A.; Lundqvist, K.; Pettersson, P.; Jaradat, O. Automated verification of AADL-specifications using UPPAAL. In Proceedings of the 2012 IEEE 14th International Symposium on High-Assurance Systems Engineering (HASE), Omaha, NE, USA, 25–27 October 2012; pp. 130–138.
47. Bruintjes, H.; Katoen, J.P.; Lesens, D. A statistical approach for timed reachability in AADL models. In Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Rio de Janeiro, Brazil, 22–25 June 2015; pp. 81–88.
48. Bao, Y.; Chen, M.; Zhu, Q.; Wei, T.; Mallet, F.; Zhou, T. Quantitative performance evaluation of uncertainty-aware hybrid AADL designs using statistical model checking. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2017**, *36*, 1989–2002. [CrossRef]