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

Ashalatha Kunnappilly, Raluca Marinescu, Cristina Seceleanu

Mälardalen University, Västerås, Sweden
(first.last)@mdh.se

Abstract. Since modern ambient assisted living solutions integrate a multitude of assisted-living functionalities, some are safety-critical, it is desirable that these systems are analyzed already at their design stage to detect possible errors. To achieve this, one needs suitable architectures that support the seamless design of the integrated assisted-living functions, as well as capabilities for the formal modeling and analysis of the architecture. In this paper, we attempt to address this need, by proposing a generic integrated ambient assisted living system architecture, consisting of sensors, data-collector, local and cloud processing schemes, and an intelligent decision support system, which can be easily extended to suite specific architecture categories. Our solution is customizable, therefore, we show three instantiations of the generic model, as simple, intermediate and complex configuration, respectively, and show how to analyze the first and third categories by model checking. Our approach starts by specifying the architecture, using an architecture description language, in our case, the Architecture Analysis and Design Language that can also account for the probabilistic behavior of such systems. 69 ecifications are semantically anchored into a formal model. To enable formal analysis, we describe the semantics of the simple and complex categories as stochastic timed automata. The former we model check exhaustively with UPPAAL, whereas for the latter we employ statistical model checking using UPPAAL SMC, the statistical extension of UPPAAL, for scalability reasons. Our work paves the way for the development formally-assured future ambient assisted living solutions.

1 Introduction

Elderly people across the world are offered enhanced care via the Ambient Assisted Living (AAL) solutions that support their independent and low-risk living. In order to facilitate the elderly support efficiently and safely, it is often required that these solutions integrate various assisted-living functionalities like health monitoring, home monitoring, fall detection, robotic platform support, communication support, etc. Such integration is extremely beneficial in safety-critical situations, for instance, the case of a fall event occurring due to low pulse, which should trigger sending timely alerts to caregivers, for immediate intervention or else the life of the elderly can be endangered. This requires timely integration of health monitoring (in this case, pulse monitoring) and fall detection functionalities. However, in literature, there are only few architectures, that address the concern of multiple-functionality integration in a timely and robust manner [32,33]. Due to their critical nature, it is beneficial that such behaviors (especially those emerging due to multiple functionality integration) are analyzed at early stages of development, for instance, at design stage, using formal techniques, to provide some formal guarantees of meeting requirements. There has been some work in this direction, however, the existing frameworks [15,44] are still in infancy and cannot be used to specify the complete AAL system architecture including its artificial intelligent algorithms, timeliness, reliability, and fault-tolerance attributes.

In this paper, we describe these shortcomings and propose an integrated architecture framework for describing AAL systems and a formal analysis framework that can be employed at the design stages of development. The integrated AAL architecture that we propose supports a range of assisted-living functionalities, like health monitoring, fall detection, reminder services, home monitoring, robotic platform support, etc. and follows the design of common AAL frameworks, with a variety of sensors, data collector unit, user interfaces, intelligent **decision support system (DSS)**, local and cloud processing, etc. Our architecture gives due importance to intelligent decision making by proposing a DSS that employs a mix of artificial intelligent (AI) techniques, like *fuzzy reasoning, rule-based reasoning (RBR) and case-based reasoning (CBR)* for effectively modelling the context space and taking the respective actions based on the current context. The system architecture and its DSS is designed as a generic model that can be customized to fit various categories of architectures, of different complexity. In this work, we show three of such instantiations of our generic model, that is, i) **a minimal configuration** that contains two sensors (pulse and fall), one user interface (a mobile phone), and a cloud controller with a simple DSS system to handle the events from both the sensors, ii) **an intermediate one** with added sensors for blood pressure monitoring, motion detection and exercise monitoring and an enhanced cloud DSS, and iii) **a complex one** comprising wider categories of health monitoring and home monitoring sensors, multiple user interfaces inclusive of robotic telepresence and vocal interactions, and a complex DSS system for handling multiple events simultaneously, and possessing both local and cloud copies for ensuring fault-tolerance via redundancy. The system architecture, its DSS, and instance models are explained in detail in Section 4.

Our contributions also include a modelling and analysis framework proposed for the design-time analysis of complex AAL systems as described earlier. The architecture design relies on the *Architecture Analysis and Design (AADL)* language in which we show the structure and communication between the components of our proposed solution. In AADL, we are able to design the architecture together with the functional and error behavior of the constituting components (Section 2.1). Once described, the architecture needs to be analyzed formally for meeting functional and quality-of-service requirements (end-to-end deadlines, fault tolerance, etc.). To enable this, we transform the architecture specifications to a formal model, in our case, the stochastic timed automata (STA) model, that can effectively capture the probabilistic behaviour of AAL components such as random component failures. We demonstrate our formal analysis via two techniques: a) **exhaustive model-checking** using the state of art model checker, UPPAAL, in case of the minimal architecture configuration (for which exhaustive verification scales) and b) **statistical model-checking** with UPPAAL SMC for analyzing the complex model instance [20]. The analysis results are described in (Section 7). Although the analysis results are not exact in case of statistical model-checking, these simulation-based methods are sometimes the only choice for reasoning of such complex cyber-physiscal systems (CPS) [16,35].

2 Preliminaries

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

2.1 The Architecture Analysis and Design Language

AADL [25] is a textual and graphical language in which one can model and analyze a real-time system's hardware and software architecture as hierarchies of components at various levels of abstraction. AADL component categories like ***Application Software*** (*Process, Data, Subprogram, Thread,* and *Thread Group*, etc.), ***Execution Platform***(*Device, Bus, Processor, Memory,* etc.) and ***System*** are used to represent the 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 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)* [28] and the *Error Annex (EA)* [22] 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* [24]. A *component library* creates a repository of component types and implementations with simple hierarchy. It can be established via two packages: (i) *Interfaces Library* comprising generic components like sensors, actuators and user-interfaces (UI), and (ii) *Controller Library* that includes the control logic. The *Reference architecture* creates a repository of components of complex hierarchy, e.g. the top-level system architecture.

2.2 Formal Notations and Tools

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

2.3 Timed Automata and Stochastic Timed Automata

A timed automaton (TA) as used in the model checker UPPAAL is a formal notation for describing real-time systems [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, $\Sigma = \Sigma_\mathrm{i} \cup \Sigma_\mathrm{o}$, and $\tau \notin \Sigma$ denotes internal or empty actions without synchronization,

V is a set of *data variables*, C is a set of *clocks*, $E \subseteq L \times B(C,V) \times A \times 2^C \times L$ is the set of *edges*, where $B(C,V)$ is the set of *guards* over C and V, that is, conjunctive formulas of clock constraints ($B(C)$), of the form $x \bowtie n$ or $x - y \bowtie n$, where $x, y \in C$, $n \in \mathbb{N}$, $\bowtie \in \{<, \leq, =, \geq, >\}$, and non-clock constraints over V ($B(V)$), and $I : L \longrightarrow B_{dc}(C)$ is a function that assigns *invariants* to locations, where $B_{dc}(C) \subseteq B(C)$ is the set of downward-closed clock constraints with $\bowtie \in \{<, \leq, =\}$. The invariants bound the time that can be spent in locations, hence ensuring progress of TA's execution. An edge from location l to location l' is denoted by $l \xrightarrow{g,a,r} l$, where g is the guard of the edge, a is an update action, and r is the clock reset set, that is, the clocks that are set to 0 over the edge. A location can be marked as *urgent* (marked with an U) or *committed* (marked with a C) indicating that the time cannot progress in such locations. The latter is a more restrictive, indicating that the next edge to be transversed needs to start from a *committed* location.

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

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

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

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

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

The delay density function (μ) over delays in $\mathbb{R}_{\geq 0}$ is either a uniform or an exponential distribution depending on whether the time in location l is bounded by an invarinat, or is unbounded, respectively. With E_{l} we denote the disjunction of guards g such that $l \xrightarrow{g,o,-} - \in E$ 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 $\{o : (l, g, o, \text{-}, \text{-}) \in E \wedge v \models g\}$.

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

2.4 UPPAAL and UPPAAL SMC

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) [13], and checked by the UPPAAL model checker. UPPAAL supports verification of liveness and safety properties [34]. 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 [20], the extension of UPPAAL for statistical model checking, provides means to formally analyze stochastic models. A model in UPPAAL SMC consists of a network of interacting STA (NSTA) that communicate via broadcast channels and shared variables. In a broadcast synchronization one sender $c!$ can synchronize with an arbitrary number of receivers $c?$. In the network, the automata repeatedly race against each other, that is, they independently and stochastically decide how much to delay before delivering the output, and what output to broadcast at that moment, with the "winner" being the component that chooses the minimum delay. In addition to the classical queries supported by UPPAAL, UPPAAL SMC also uses an extension of weighted metric temporal logic (WMTL) [19] 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, caregivers. Accordingly, the architecture supports a variety of components like sensors, a data collector unit

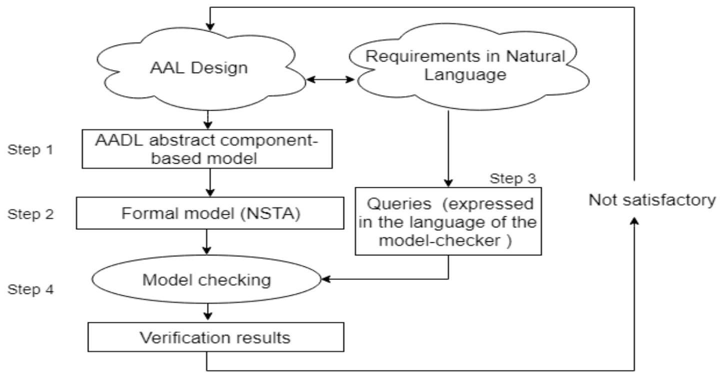

AAL Design
Requirements in Natural Language
Step 1
AADL abstract component-based model
Step 2
Formal model (NSTA)
Step 3
Queries (expressed in the language of the model-checker)
Not satisfactory
Step 4
Model checking
Verification results

Fig. 1: Methodology overview.

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 Fig. 1, the framework is composed of the following steps:

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

This step focuses on specifying the architecture using an architecture description language. In our case, we have chosen AADL due to its rich semantics and suitability to model real-time embedded systems. In our approach, we demonstrate the modeling of AAL systems as abstract components and show how it can be extended to suit the specific instantiations (from simpler to more complex configurations, as shown in Section 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 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. Thus, Step 3 is formulated as follows:

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

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

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

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

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 vary in their degree of complexity with respect to integrated functionalities.

The generic AAL system architecture is presented in Fig.2, and follows the architecture of many commercial AAL systems with various sensors, a data collector, DSS, security and privacy, database (DB) systems, user interfaces (UI), and cloud computing support. This architecture can act as a base for the development of many integrated AAL system architectures. We classify the sensors in the AAL environment as follows:

- Wearable sensors that send information as data (W_data), e.g., sensors measuring health parameters like pulse, ECG, etc. They are represented by Sensor_A category in Fig 2;
- Non-wearable sensors measuring ambient parameters and health parameters (NW_data), e.g., camera sensors, motion sensors, etc., represented by Sensor_B category;
- Wearable sensors that detect events (W_event), e.g., fall sensors, marked as Sensor_C category;
- Non-wearable sensors detecting events (NW_event), e.g., fire sensors, denoted by Sensor_D category.

A particular instantiation of the generic architecture can contain n sensors of each category, respectively, $n \in N$. As depicted in Fig.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

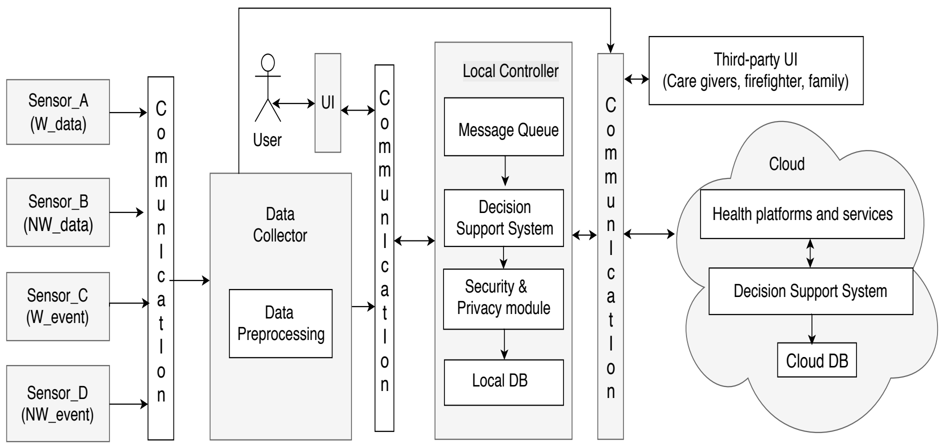


Fig. 2: The generic AAL system architecture.

that when the DSS processes the first element in the queue, it processes the message with the highest priority. Our architecture has both local and cloud-based processing in order to ensure fault tolerance with respect to the DSS. The components of the architecture can interact via various communication protocols.

The crux of our AAL system is the **intelligent context-aware DSS**, shown in Fig.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.

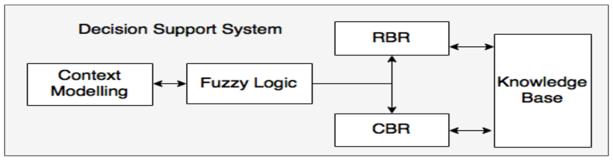


Fig. 3: The DSS architecture

Our DSS architecture is inspired by the work of Zhou et al. [46], where the authors have proposed a context-aware, CBR-based ambient-intelligence system for AAL applications. CBR reasoning works very well in scenarios that are not specific and need to adapt accordingly with inputs. For instance, CBR reasoning is suited in a clinical decision support system that prescribes medicines/treatment, where the treatment, prescription and medicine dosage vary depending upon individual patients. CBR is an attractive choice due to its reasoning technique resembling more of human problem-solving competence, (i.e., trying to reason out a new scenario by looking at the similar solved cases in the past and

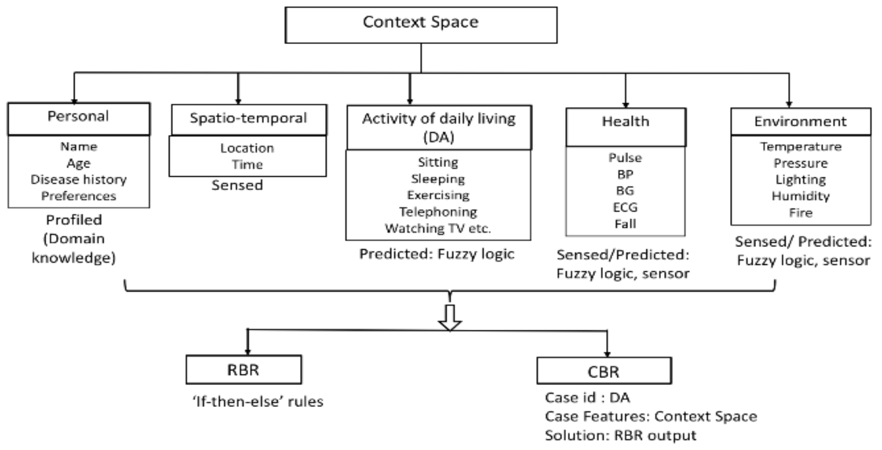


Fig. 4: Internals of the DSS architecture (List of AI techniques)

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

The DSS triggers the various AI algorithms based on a change in *context* [46]. The context-modeling (CM) and the usage of different AI algorithms are depicted in Fig. 4. As indicated, CM module identifies the context space based on: (i) the personal profile of the user, e.g., gender, age, disease history etc., (ii) the activity of daily living (DA) performed by the user, e.g., exercising, sleeping etc., (iii) spatio-temporal properties, like time, location of the user, etc., (iv) environmental, e.g., temperature, pressure, fire, etc., and (v) health parameters, for instance, like blood pressure, pulse, etc. Each of these context-space com-

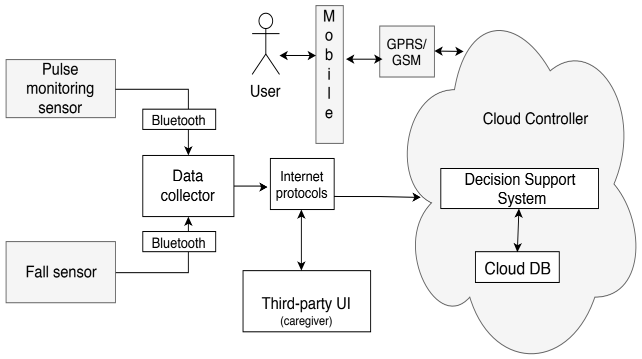


Fig. 5: Category 1: A minimal configuration

ponents can be associated with one of the three properties - *sensed, profiled* or *predicted*. *Sensed* contexts are those directly derived from sensor values. However, predicted contexts correspond to the output resulting from further analysis of sensed inputs, e.g., activity-recognition. *Profiled* values are usually descriptive and remain unchanged.

In our DSS, fuzzy reasoning is used for detecting DA [40], and also for determining sensor-data deviations [1]. To take decisions in various situations, we employ RBR first, CBR as second paradigm, i.e., upon a change in context, the RBR triggers first and checks if there exists a rule to handle that particular context, if not, it allows the CBR system to tackle the context based on its learning from previous scenarios. Developing an efficient case base, case matching and formulating the adaptation rules are the most complex aspects of a CBR system. In our system, each time an RBR outputs a rule, we save it as a *case* in the CBR system with the *case-id* represented by the DA of the user, the *context space* represented by the case features, and the triggered *rule* represented by the solution for a particular case. The KB stores the context, rules, and cases. The internal structure of the DSS is represented in Fig.4. An example scenario of the DSS reasoning employing different AI techniques is presented in detail in Listing 1.1 of Section 6.2.

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

– **Category 1: A minimal configuration** - The minimum configuration architecture consists of the following modules: Two sensors (a fall sensor and a pulse monitoring sensor), a mobile phone UI, and cloud controller with third-party UI and DSS system with a minimum context-space information including the health data (pulse and fall) and DA. The simplified DSS em-

[1] In order to reduce the complexity of our analysis, we have not explicitly modeled the DA detection using fuzzy logic and has often assumed that the user's DA is known in various scenarios.

ploys only RBR with fuzzy logic as AI techniques. The minimal configuration is shown in Fig. 5.

- **Category 2: An intermediate configuration** - This instantiation (see Fig. 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.

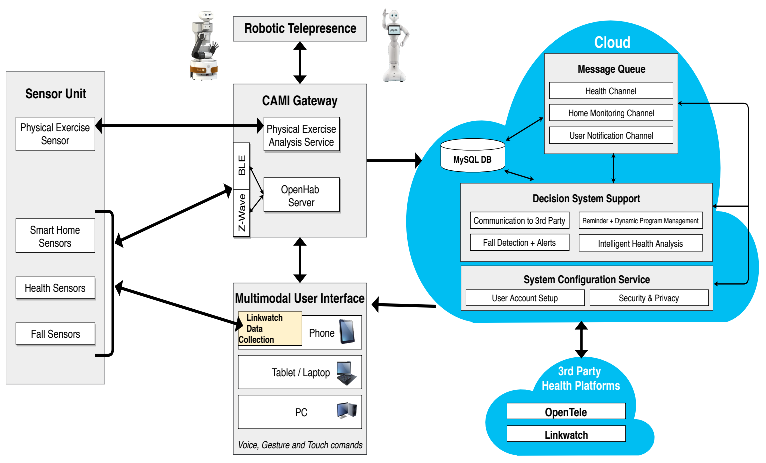


Fig. 6: Category 2: An intermediate configuration

- **Category 3: A complex configuration** - In this category, we present the most complex version, the CAMI AAL architecture [33] derived from our generic model, and represented in Fig. 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 [10], Fibaro temperature and motion sensor FGMS-001 [2], Fitbit bracelet [3], Vibby fall detection sensor [11], etc.), data collector, local controller (EXYS9200-SNG [1] referred as *CAMI gateway*), the *CAMI cloud*, and third party health platforms like *Open Tele* [5] and [4]. There is a set of user interfaces (UI) in CAMI, including robotic platforms (TIAGo [9] and Pepper [7]), 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 [8]. 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.

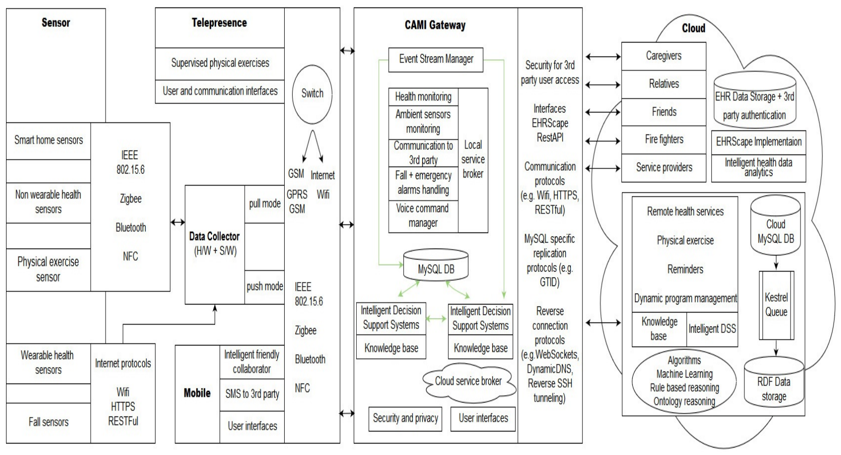


Fig. 7: Category 3: A complex configuration: The CAMI AAL System Architecture [33]

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

4.1 Use Case Scenarios and System Requirements

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

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

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

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

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

– *Scenario 3 - Home-monitoring functionalities*: Jim forgets to switch off the cooker after cooking his dinner, which results in a fire in the house. The fire detection sensor of CAMI detects the fire and the system alerts the firefighters of the fire incident in Jim's house.
– *Scenario 4 - Combining various functionalities in case of multiple events occurring together*: Jim is cooking his breakfast. He suddenly feels dizzy and falls. The gas-based cooker is still on, and eventually starts a fire in Jim's house. In this case, the CAMI system detects the simultaneously occurring events, and alerts the firefighter and caregiver of both the events. As a result, the firefighters and caregivers can immediately start the rescue without waiting for alarm confirmations, avoiding potentially dangerous consequences [32]. Further, if there are any health parameter variations detected for Jim along with the fall (for instance, a low pulse), the fall event can be associated with the low pulse, and the caregiver notified accordingly, which can help in further diagnosis.

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

Requirements of the minimal architecture model (Category 1):

– $\mathbf{R1_{Arch1}}$: 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.
– $\mathbf{R2_{Arch1}}$: 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.

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.

– $\mathbf{R1_{CAMI}}$: 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.

– $\mathbf{R2_{CAMI}}$: 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.

– $\mathbf{R3_{CAMI}}$: 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.

– $\mathbf{R4_{CAMI}}$: 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.

– $\mathbf{R5_{CAMI}}$: 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.

– $\mathbf{R6_{CAMI}}$: 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 Fig. 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.

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[2]. 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 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 $\mathrm{R3}_{\mathrm{CAMI}}$ (Scenario 1), described in Section 2.1. The RBR component type, implementation, BA, and EA are shown in Listing 1.1. 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 16-20) defines the data sub-components like the fuzzy data output, personal information and daily activity of the user, which forms the **context-space** of Scenario 1.

In the BA (Lines 22-28), *Waiting* represents the initial state where the component waits for an input from the pulse sensor. In the *Operational* state, the system monitors the **fuzzy logic** output to identify any pulse variations. The fuzzy reasoning is not shown in Listing 1.1 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

[2] 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.

encoded as a rule in the BA depicted in Listing 1.1. 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 Fig. 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. For a full description of the RBR model in AADL, the user can refer to the Appendix A.

Listing 1.1: An excerpt from the RBR component in AADL for CAMI

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

```
41  t1: Waiting -[PF]->Failed_Permanent
42  t2: Waiting -[TF]->Failed_Transient;
43  t3: Failed_Transient -[Reset]-> {LReset with 0.9,
44  Failed_Permanent with 0.1};
45  t4: LReset-[]->{Waiting with 0.8, Failed_Permanent with 0.2}
46  properties
47  EMV2::DurationDistribution => [Duration => 1s..2s; applies to Reset;
48  EMV2::OccurrenceDistribution =>[ProbabilityValue => 0.9;
49  Distribution => Fixed;] applies to Reset;
```

AAL Composite Components: A CC is defined in a similar way as that of AC, except that its BA is not explicitly defined (We assume that the behaviour of the CC is already encoded by its sub-components). Also, the EA definition of CC shows the failure behaviour of its sub-components. In Listing 1.2, 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 1.2: An excerpt from the DSS component in AADL for CAMI

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

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

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

- $Comp_{\text{type}}$ is defined as a tuple: $Comp_{\text{type}} = \langle Features, Flow_{\text{spec}}, Prop \rangle$, where:
  - $Features = IN_{\text{p}} \cup OUT_{\text{p}}$, where IN_{p}, OUT_{p} represent the sets of *input ports* and *output ports* respectively, and IN_{p}, $OUT_{\text{p}} \in \{$*data-ports*, *event-ports*, *event-data-ports*$\}$;
  - $Flow_{\text{spec}} = \langle Flow_{\text{so}}, Flow_{\text{p}}, Flow_{\text{si}} \rangle$, where $Flow_{\text{so}}$, $Flow_{\text{p}}$, $Flow_{\text{si}}$ represent flow sources, flow paths and flow sinks respectively. Let $F_{\text{s0}} : Flow_{\text{so}} \to OUT_{\text{p}}$ be a function that associates certain OUT_{p} to $Flow_{\text{so}}$ with $Flow_{\text{so}} \subseteq OUT_{\text{p}}$, $F_{\text{p}} : Flow_{\text{p}} \to OUT_{\text{p}} \times IN_{\text{p}}$ be a function that associates and an input and an output to a flow, and $F_{\text{si}} : Flow_{\text{si}} \to IN_{\text{p}}$ be a function that associates certain IN_{p} to $Flow_{\text{si}}$, with $Flow_{\text{si}} \subseteq IN_{\text{p}}$. For instance, in our AAL architecture, we can define $Flow_{\text{spec}}$ for fall events by defining the output port of the fall sensor as $Flow_{\text{so}}$, the input port of the cloud DSS as $Flow_{\text{si}}$, and the input and output ports of all the intermediate components defining the Flow_{p};

  - $Prop$ is the set of associated properties of the component, like *Deployment*, *Communication*, *Timing*, *Thread-related properties*, etc. [24]. In this work, we only consider a subset of *Timing*, *Thread-related properties*, and *user- defined properties*, that are represented as follows: $Prop = \{T_{\mathrm{p}}, T_{\mathrm{e}}, Dispatch\ protocol, event\_gen\_dist, failure\_recovery\_dist\}$ where T_{p} and T_{e} represent the period and execution-time of the component, respectively, T_{p}, $T_{\mathrm{e}} \in$ *Timing properties*, $Dispatch\ protocol \in \{P, AP\}$[3], where P represents a Periodic and AP represents an Aperiodic protocol, and $P, AP \in$ *Thread-related properties*, and $event\_gen\_dist, failure\_recovery\_dist \in$ *user- defined properties* represent the set of user-defined properties used for specifying the occurrence distribution of aperiodic events and failure recovery, respectively.

– $Comp_{\mathrm{imp}}$ is defined as $Comp_{\mathrm{imp}} = \langle SC, P_{\mathrm{t}}, Con, MSM, Flow_{\mathrm{imp}}, ETF\rangle$, where:
  - SC represents the set of sub-components of the system with port interfaces (SC_{i}) and without port interfaces (SC_{Data}), i.e., $SC = SC_{\mathrm{Data}} \cup SC_{\mathrm{i}}$;
  - P_{t} denotes the set of *Prototypes* used to define SC via $Fp : P_{\mathrm{t}} \rightarrow SC_{\mathrm{i}} \times SC_{\mathrm{Data}}$, a function that associates SC to a P_{t}, respectively;
  - Con represents the set of connections. $F_{\mathrm{con}} : Con \rightarrow Features$ is a function that assigns $Features$ to Con;
  - MSM is the mode state machine that is modeled by a tuple, as follows: $MSM = \langle M_{\mathrm{s}}, \rightarrow\rangle$, where M_{s} is the set of states, and $\rightarrow \subseteq M_{\mathrm{s}} \times ev \times M_{\mathrm{s}}$ is the transition relation (with ev being the set of events, such that $Fe :$ *event-ports* $\rightarrow ev$, *event-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;
  - $Flow_{\mathrm{imp}}$ are the flow implementations, represented as $Flow_{\mathrm{imp}} : SC \rightarrow Flow_{\mathrm{spec}}$;
  - 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: $EA = \langle E_{\mathrm{flows}}, E_{\mathrm{beh}}, E_{\mathrm{prop}}\rangle$, where:
  - E_{flows} denotes the error flows, $E_{\mathrm{flows}} = \langle E_{\mathrm{pp}}, Err_{\mathrm{so}}, Err_{\mathrm{p}}, Err_{\mathrm{si}}\rangle$, where E_{pp} describes error propagations, and Err_{so}, Err_{p}, Err_{si} represents error sources, error paths, and error sinks, respectively; $F_{\mathrm{e1}} : Err_{\mathrm{so}} \rightarrow OUT_{\mathrm{p}}$ is a function that associates certain output ports with error sources, $F_{\mathrm{e2}} : Err_{\mathrm{p}} \rightarrow (IN_{\mathrm{p}}, OUT_{\mathrm{p}})$ is a function that associates input and output ports via Err_{p}, $F_{\mathrm{e3}} : Err_{\mathrm{si}} \rightarrow IN_{\mathrm{p}}$ is a function that assigns certain input ports as error sinks;

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

- E_{beh} represents error behavior, $E_{\mathrm{beh}} = \langle E_{\mathrm{s}}, \rightarrow_{\mathrm{e}}, E_{\mathrm{e}}, EM_{\mathrm{Comp}} \rangle$, where E_{s} represents the set of error states, $\rightarrow_{\mathrm{e}}$ denotes an error transition relation, $\rightarrow_{\mathrm{e}} \subseteq E_{\mathrm{s}} \times E_{\mathrm{e}} \times E_{\mathrm{s}}$, 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, $s_{\mathrm{e}}, s_{\mathrm{e}}' \in E_{\mathrm{s}}$, and $\rightarrow_{\mathrm{e}}$ the transition between them due to an error event $e_{\mathrm{e}} \in E_{\mathrm{e}}$, then $s_{\mathrm{e}} \xrightarrow{e_{\mathrm{e}}}_{\mathrm{e}} s_{\mathrm{e}}'$. We represent initial state as $s_{0\mathrm{e}} \in E_{\mathrm{s}}$. $F_{E_{\mathrm{pp}}} : Epp \rightarrow (IN_{\mathrm{p}}, OUT_{\mathrm{p}})$ 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* ($Occur_{\mathrm{dist}}$), which aid in our error analysis, thus $E_{\mathrm{prop}} = \{Dur_{\mathrm{dist}}, Occur_{\mathrm{dist}}\}$.

– The Behaviour Annex, BA is defined as: $BA = \langle B_{\mathrm{v}}, B_{\mathrm{s}}, \rightarrow_{\mathrm{b}} \rangle$, where B_{v}, B_{s}, represent the set of variables, and the states of BA, respectively and $\rightarrow_{\mathrm{b}}$ is a BA transition relation. Let s_{b} and s'_{b} be two states of BA, $s_{\mathrm{b}}, s'_{\mathrm{b}} \in B_{\mathrm{s}}$, and $\rightarrow_{\mathrm{b}}$ the transition between them, $\rightarrow_{\mathrm{b}} \subseteq B_{\mathrm{s}} \times B_{\mathrm{v}} \times SC_{\mathrm{Data}} \times B_{\mathrm{s}}$, with SC_{Data} being the set of data subcomponents. We denote by $s_{0\mathrm{b}} \in B_{\mathrm{s}}$ the initial state of a BA path.

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

– $AC \in AADL_{\mathrm{Comp}}$, where $Comp_{\mathrm{ImplAC}} = \{SC_{\mathrm{Data}}\}$, $EA_{\mathrm{AC}} \neq \emptyset$, where $E_{\mathrm{beh}} \in EA_{\mathrm{AC}} = \{E_{\mathrm{s}}, \rightarrow_{\mathrm{e}}, E_{\mathrm{e}}\}$, $BA_{\mathrm{AC}} \neq \emptyset$,
– $CC \in AADL_{\mathrm{Comp}}$, where $Comp_{\mathrm{ImplCC}} = \{P_{\mathrm{t}}, SC_{\mathrm{i}}, SC_{\mathrm{Data}}, Con, MSM, Flow_{\mathrm{imp}}, ETF\}$, $EA_{\mathrm{CC}} \neq \emptyset$, where $E_{\mathrm{beh}} \in EA_{\mathrm{CC}} = \{EM_{\mathrm{Comp}}\}$, $BA_{\mathrm{CC}} = \emptyset$. 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 1.1). In formal semantics, we define it as follows:

$$RBR_{\mathrm{AADL}} = \langle Comp_{\mathrm{type\ RBR}}, Comp_{\mathrm{imp\ RBR}}, EA_{\mathrm{RBR}}, BA_{\mathrm{RBR}}, \rangle \qquad (4)$$

where the elements are defined as follows:

– $Comp_{\mathrm{type\ RBR}} = \langle Features_{\mathrm{RBR}}, Flow_{\mathrm{spec\ RBR}}, Prop_{\mathrm{RBR}} \rangle$, with:
  - $Features_{\mathrm{RBR}} = IN_{\mathrm{p}} \cup OUT_{\mathrm{p}}$, and IN_{p}, $OUT_{\mathrm{p}} \in \{$ *event-data-ports*$\}$,
  - $Flow_{\mathrm{spec\ RBR}} = \langle Flow_{\mathrm{p}} \rangle$,
  - $Prop_{\mathrm{RBR}} = \{T_{\mathrm{e}}, AP\}$.

– $Comp_{\mathrm{imp\ RBR}} = \langle SC_{\mathrm{DataRBR}} \rangle$
– $EA_{\mathrm{RBR}} = \{Err_{\mathrm{p}}, E_{\mathrm{s}}, \rightarrow_{\mathrm{e}}, E_{\mathrm{e}}, Dur_{\mathrm{dist}}, Occur_{\mathrm{dist}}\}$
– $BA_{\mathrm{RBR}} = \{B_{\mathrm{s}}, \rightarrow_{\mathrm{b}}\}$

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

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

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.

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

- *The* $\boldsymbol{AC_{iSTA}}$ *is defined according to a template STA (see Fig. 8) with* $L \in \{Idle, Op, Fail, start, stop\}$, $l_0 = Idle$*, Op corresponds to the Operational state of the RBR, start, stop represent the locations to initiate the synchronizations with* AC_{aSTA} *and* $E = \{Idle \longrightarrow start, start \longrightarrow Op, Op \longrightarrow stop, stop \longrightarrow Idle, Op \longrightarrow Fail, Fail \longrightarrow Idle\}$*. This template is annotated with the following information:*
  - $V = out\_port \cup in\_port \cup \{PF, TF\} \cup SC_{Data}$ *, where out_port and in_port represent the set of output and input ports* $\in$ *{data-ports, event-ports, event-data-ports}, respectively, and the Boolean variables,* PF, TF*, represent the error events associated with the transient failure and permanent failure of AC, plus the variable associated with* $SC_{Data} \in Comp\_imp$*;*

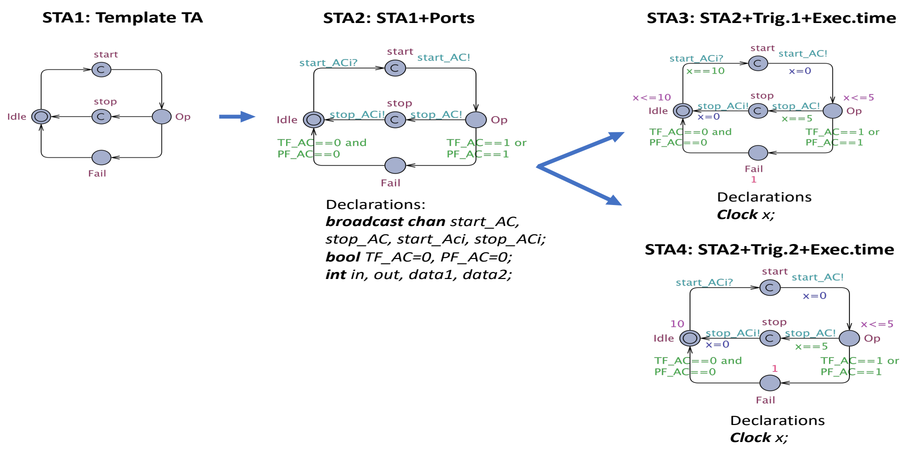


Fig. 8: Step-by-step formulation of AC_{iSTA}

- *$C = \{x\}$ 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-data-ports, event-ports$\}$, that is, channels $start\_AC!, stop\_AC!$, and the synchronization channels for the interface of the corresponding CC, that is, $start\_ACi?, stop\_ACi!$ and the reset actions on x;*
- *$E = \{Idle \xrightarrow{start\_ACi? \wedge x == T_p} start, start \xrightarrow{start\_AC!, x=0} Op,$ $Op \xrightarrow{TF\_AC==1 \vee PF\_AC==1} Fail, Op \xrightarrow{x==T_e, stop\_AC!} stop,$ $stop \xrightarrow{stop\_ACi!, x=0} Idle, Fail \xrightarrow{TF\_AC==0 \wedge PF\_AC==0} Idle,$ $Fail \xrightarrow{TF\_AC==1 \wedge PF\_AC==1} Fail\}$, where E is defined by the template populated with A and guards that ensure the correctness of transitions.*
- *$I(Idle)=(x \leq T_p)$, if the dispatch protocol associated with AC is periodic, and $I(Op) = x \leq T_e$, where T_p and T_e represent the period and execution-time of AC;*
- *$P(Idle) = \mu_1$, and $P(Fail) = \mu_2$, where $P(Idle) = \mu_1$ represents the occurrence distribution of aperiodic event (if the dispatch protocol associated with AC is aperiodic), and $P(Fail) = \mu_2$ represents the probability of leaving location $Fail$;*

– *The $\boldsymbol{AC_{aSTA}}$ is created in a similar way with:*
  - *$L = \{Wait, Op, TrF, PrF, Fail\_ep, LReset, L1, L2\}, l0 = Wait$, where L comprises the set of states in EA and BA (Wait, Operational (Op), Transient Failure (TrF), Permanent Failure (PrF), Failed due to error propagation (Fail_ep), and reset location (LReset), plus additional committed locations $(L1, L2)$ that ensure that receiving is deterministic in UPPAAL SMC;*

- $A = \{start\_AC?, stop\_AC?\} \cup \{action_{BA,EA}(), TF = 0, TF\_AC = 1, PF\_AC = 1, reset\_AC = 0, reset\_AC = 1, err\_pAC = 0, err\_pAC = 1, err\_p = 1, y = 0\}$, *where A is composed of the actions defined in BA and EA ($action_{BA,EA()}$), plus the synchronizations channels to concord with AC_{iSTA} ($start\_AC?, stop\_AC?$), and the reset of clock y;*
- $V = \{PF\_AC, TF\_AC, reset\_AC, err\_pAC\}$, *where V consists of the set of error events defined in the EA, that is, PF_AC : Permanent Failure of AC, TF_AC: Transient Failure of AC, reset_AC: Reset of AC, err_pAC: error propagation of AC;*
- $C = \{y\}$ *is the clock that measures the time elapsed for reset action of a particular component;*
- $E = \{Wait \xrightarrow{start\_AC?} L1, L1 \xrightarrow{TF\_AC=1, err\_pAC=1} TrF, L1 \xrightarrow{PF\_AC=1, err\_pAC=1} PrF, L1 \rightarrow L2, L2 \rightarrow Op, Op \xrightarrow{stop\_AC?, action_{BA}()} Wait, TrF \xrightarrow{reset\_AC=1, y=0} LReset,$
  $TrF \xrightarrow{PF\_AC=1, err\_pAC=1, reset\_AC=0} PrF,$
  $LReset \xrightarrow{TF\_RBR=0, err\_pAC=0, reset\_AC=0} Wait,$
  $LReset \xrightarrow{PF\_AC=1, err\_pAC=1, reset\_AC=0} PrF,$
  $Wait \xrightarrow{err\_p==1} Fail\_ep\}$, *where E consists of the transitions in EA, BA and those between $L1$ and $L2$;*
- $I(LReset) = (y \leq Dur_{dist(Reset)})$;
- $P(Wait) = \mu$, *that is the occurrence-distribution of $Wait$;*
- $L1 \xrightarrow{\gamma 1} L2$, $L1 \xrightarrow{\gamma 2} TrF$, $L1 \xrightarrow{\gamma 3} PrF$, *where $\gamma 1, \gamma 2, \gamma 3$, are defined according to the occurrence-distribution of the error events.* □

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

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

represent the error events associated with the transient failure and permanent failure of CC, plus the variable associated with $SC_{Data} \in Comp\_imp$*;*
  - $C = \{x\}$ *if* $T_p \neq \emptyset$*;*
  - *A is defined based on the updates defined by* MSM*, the updates defined by* $Flow_{imp}$*, the synchronizations defined by* Con*, the synchronization with* CC_{aSTA}*,* AC_{aSTA}*, and in case C is not void, we add the clock reset of the clock(s) in C;*
  - $I(Wait)$=$(x \leq T_p)$ *if* $T_p \neq \emptyset$*;*
  - $P(l) = \mu$*, where* $l \in L$ *and* μ *is defined by* $Prop$*.*
- $\boldsymbol{CC_{aSTA}}$ *is defined as follows:*
  - $L = E_s \in EA$*,* $l_0 = s_{0e} \in E_s$*, where* E_s *is the set of states of EA;*
  - $E = \rightarrow_e$*;*
  - $A = \{TF\_CC = 1, TF\_CC = 0, PF\_CC = 1\}$*;*
  - *V is represented by the global variables defined in* CC_{iSTA}*;*
  - $C = \emptyset$*;*
  - $P(l) = \mu$*, where* $l \in L$ *and* μ *is defined by* $Occur_{dist} \in E_{prop}$*.*

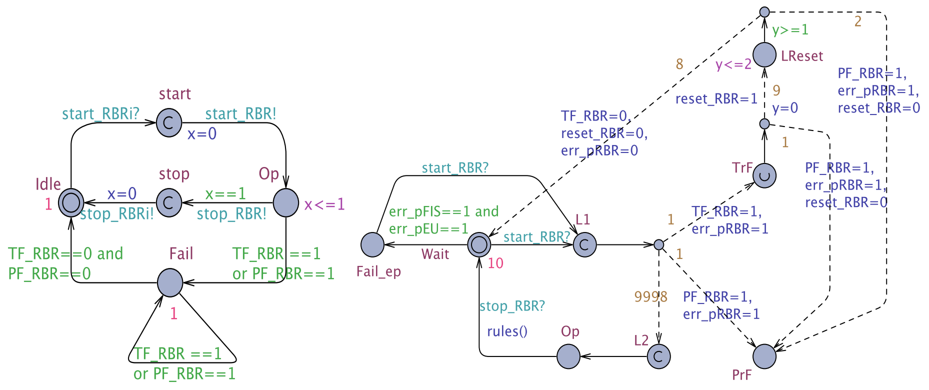
All the other CC elements are transformed based on the encoding EA of AC.

□

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

The RBR_{AADL} defined by Eq.(4), is mapped into an NSTA (RBR_{NSTA}) following the Definition 1: RBR_{NSTA}=$RBR_{\text{iSTA}}||RBR_{\text{aSTA}}$ (Fig. 9), where RBR_{iSTA} is the so-called "Interface STA" of RBR which corresponds to $Comp_{\text{type RBR}}$ and $Comp_{\text{impl RBR}}$, whereas RBR_{aSTA} is the "Annex STA" of RBR that encodes its EA and BA.

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



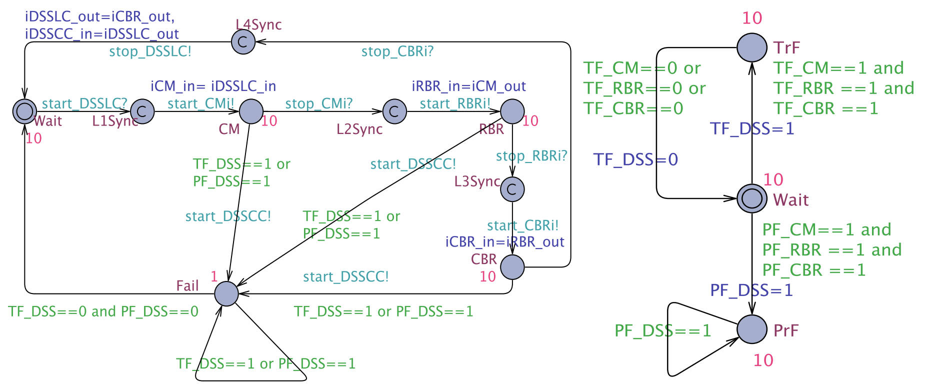
(a) Interface STA (RBR_{iSTA}) (b) Annex STA (RBR_{aSTA})

Fig. 9: The STA for the RBR

- $C = \{y\}$
- $E = \{Wait \xrightarrow{start\_RBR?} L1, L1 \xrightarrow{TF\_RBR=1,err\_pRBR=1} TrF, L1 \xrightarrow{PF\_RBR=1,err\_pRBR=1} PrF, L1 \rightarrow L2, L2 \rightarrow Op, Op \xrightarrow{stop\_RBR?,rules()} Lsync, Lsync \xrightarrow{stop\_RBRi!} Wait, TrF \xrightarrow{reset\_RBR=1,y=0} LReset,$ $TrF \xrightarrow{PF\_RBR=1,err\_pRBR=1,reset\_RBR=0} PrF,$ $LReset \xrightarrow{TF\_RBR=0,err\_pRBR=0,reset\_RBR=0} Wait,$ $LReset \xrightarrow{PF\_RBR=1,err\_pRBR=1,reset\_RBR=0} PrF, Wait \xrightarrow{err\_p==1} Fail\_ep\}$
- $I(LReset) = y \leq 2$
- $P(Wait) = 10$, 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 1.2, and represented by Eq.(5), is mapped into an NSTA: $DSS_{\text{AADL}} \rightsquigarrow DSS_{\text{NSTA}}{=}DSS_{\text{iSTA}}||DSS_{\text{aSTA}}$ (Fig. 10), where DSS_{iSTA} is the so-called "Interface STA" of DSS, which corresponds to $Comp_{\text{type DSS}}$ and $Comp_{\text{impl DSS}}$, whereas DSS_{aSTA} is the "Annex STA" that encodes the EA of CC.

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



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

Fig. 10: The STA for the DSS

$RBR \xrightarrow{stop\_RBRi?} L3Sync, L3Sync \xrightarrow{start\_CBRi!, iCBR\_in=iRBR\_out} CBR,$
$CBR \xrightarrow{stop\_CBRi?} L4Sync, L4Sync$
$\xrightarrow{stop\_DSSLC!, iDSSLC\_out=iCBR\_out, iDSSCC\_in=iDSSLC\_out} Wait, CM$
$\xrightarrow{(TF\_DSS=1 \vee PF\_DSS=1), start\_DSSCC!} Fail, RBR$
$\xrightarrow{(TF\_DSS=1 \vee PF\_DSS=1), start\_DSSCC!} Fail, CBR$
$\xrightarrow{(TF\_DSS=1 \vee PF\_DSS=1), start\_DSSCC!} Fail, Fail$
$\xrightarrow{(TF\_DSS==1 \vee PF\_DSS==1)} Fail, Fail \xrightarrow{(TF\_DSS==0 \wedge PF\_DSS==0)} Wait\}$

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

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

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

In addition to the above description, for the reader to have a deeper understanding of modeling the AI algorithms in the respective STA, we show an excerpt of the variable declarations and functions encoding that we have used to describe our DSS AI algorithms in Listing 1.3. We show the context modeling, fuzzy reasoning and RBR in the following and also show how the successful RBR outputs are stored as cases for CBR.

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

Listing 1.3: DSS model in STA in detail

```
---Context modeling---
typedef struct{
int user_name; //1 Jim
int age; //Age =65 years
int disease_history;  //3-Heart disease
}user_profile;
user_profile up;
typedef struct{
int position;
//1= inside home, 0 -outiside home
}stemporal_properties;
typedef struct{
int pulse;
int fall_w;
int fall_c;
}health_parameters;
typedef int uADL;user_profile profile;
uADL ADL; //2-exercising , 1-resting
stemporal_properties s_temp;
health_parameters health;
ambient_parameters ambient;
typedef struct{
user_profile profile;
uADL ADL;
stemporal_properties s_temp;
health_parameters health;
ambient_parameters ambient;
}context_model;

---Fuzzy Logic Reasoning---
void fuzzify()
{
if(iFIS_in.data_val>=55 and iFIS_in.data_val<=135)
{FIS_out.health.pulse=2; }
else if(iFIS_in.data_val>=40 and iFIS_in.data_val<=70)
{FIS_out.health.pulse=1;}
else if (iFIS_in.data_val<=300 and iFIS_in.data_val>=110)
{FIS_out.health.pulse=3;}
FIS_out.health.pulse=fuzzyout_pulse;
FIS_out.profile.user_name=upro.profile.user_name;
FIS_out.profile.age=upro.profile.age;
FIS_out.profile.disease_history=upro.profile.disease_history;
FIS_out.ADL =upro.ADL;
FIS_out.s_temp.position=upro.s_temp.position;
}
void update_contextEU()
{
 FIS_outsave.ambient.fire= EU_out.ambient.fire;
 FIS_outsave.health.fall_c= EU_out.health.fall_c;
 FIS_outsave.health.fall_w= EU_out.health.fall_w;
}

---RBR ---
void rules()
if ((iRBR_in.health.fall_w==1  or iRBR_in.health.fall_c==1)
and iRBR_in.ambient.fire==1)
{rule.notifications_caregiver=2;
rule.notifications_firefighter=2;}
else if (iRBR_in.health.pulse ==3 and iRBR_in.ADL==1
and iRBR_in.profile.disease_history==3)
{rule.notifications_caregiver=1;}

else if (iRBR_in.health.pulse==1 and iRBR_in.ADL==1
and iRBR_in.profile.disease_history==3)
{rule.notifications_caregiver=3;
}
else if (iRBR_in.ambient.fire==1)
{rule.notifications_firefighter=1;}
else if (iRBR_in.health.fall_c==1 or
iRBR_in.health.fall_w==1)
{rule.notifications_caregiver=7;}
RBR_o.case =upro.ADL;
RBR_o.case_features =iCM_out;
RBR_o.rule =rule;
}
```

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

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.

Exhaustive verification of the minimum configuration using UPPAAL. The results of the exhaustive verification of the minimum configuration architecture using UPPAAL model checker are tabulated in Table 1. To check that our system meets its requirements, we employ a monitor STA that monitors the sensor values, the respective DSS output, and the corresponding clock. The monitor automaton for $R1_{\text{Arch1}}$ is shown in Fig. 11. As described, we start the monitoring clock *s1* when the pulse sensor produces the data, marked by transition to *L2* triggered by the synchronization channel and we stop the clock when a decision is produced by the cloud DSS. Similar monitors have been employed for $R2_{\text{Arch1}}$. We have used queries of the form *A leads to B* for our analysis

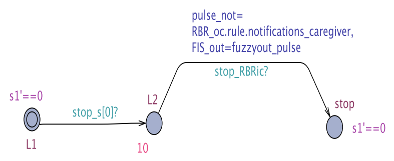


Fig. 11: The monitor automaton for requirement $R1_{\text{Arch1}}$.

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

Table 1: UPPAAL analysis results for the minimum configuration architecture

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.

Statistical Verification of the CAMI architecture using UPPAAL SMC. In case of CAMI architecture, which is the most complex instantiation of our proposed generic architecture, exhaustive verification does not scale and hence we chose to verify the CAMI system requirements using UPPAAL SMC [20], the statistical extension of UPPAAL model checker to perform probabilistic analysis. To verify the functional requirements, we employ monitor STA to monitor the sensor values, the respective DSS output and the corresponding clock. For instance, an example of monitor STA for $R1_{\text{CAMI}}$ is given in Fig. 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$R5_{\text{CAMI}}$.

Req.	Query	Result	Runs
R1$_{\text{CAMI}}$	$Pr[<= 1000]([]((M\_fire.fire\_alarm == 1)$ $imply\ (se\_nw.fire == 1 \text{ and } M\_fire.s1 <= 20)))$	Pr [0.99975,1] confidence 0.998	3868
	$Pr[<= 1000](<> (M\_fire.fire\_alarm == 1))$	Pr [0.99975,1] confidence 0.998	4901
R2$_{\text{CAMI}}$	$Pr[<= 1000]([]((M\_fall.fall\_not == 7)$ $imply\ ((se\_w.fall == 1 \text{ or } sd\_nw.data\_val == 1)$ $and(M\_fall.s1 <= 20))))$	Pr [0.99975,1] confidence 0.998	3868
	$Pr[<= 1000](<> (M\_fire.fire\_alarm == 1))$	Pr [0.99975,1] confidence 0.998	4901
R3$_{\text{CAMI}}$	$Pr[<= 1000]([]((M\_pulse.pulse\_not == 3)$ $imply\ (110 <= sd\_w.data\_val <= 300\ and$ $M\_pulse.FIS\_out == 3\ and\ ADL == 1\ and$ $upro.disease\_history == 3 \text{ and } M\_pulse.s1 <= 20))$	Pr [0.99975,1] confidence 0.998	3868
	$Pr[<= 1000](<> (M\_pulse.pulse\_not == 3))$	Pr [0.99975,1] confidence 0.998	3868
R4$_{\text{CAMI}}$	$Pr[<= 1000]([](\ M\_firefall.fire\_not == 2\ and$ $M\_firefall.fall\_not == 2\ imply$ $((se\_w.fall == 1 \text{ or } sd\_nw.data\_val == 1)\ and$ $se\_nw.fire == 1 \text{ and } M\_firefall.s1 <= 20))$	Pr [0.99975,1] confidence 0.998	3868
	$Pr[<= 1000](<> (Pr[<= 100](<> (M\_firefall.$ $fall\_not == 2 \text{ and } M\_firefall.fire\_not == 2))$	Pr [0.99975,1] confidence 0.998	7905
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

Table 2: UPPAAL SMC Analysis Results of CAMI.

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

Discussion. The approach presented in this paper paves the way for the development of formally assured future intelligent AAL solutions that integrate multiple functionalities. Our approach can be applied at earlier design stages

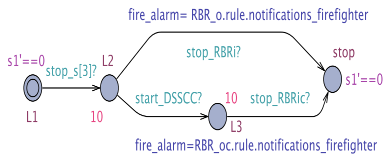

fire_alarm= RBR_o.rule.notifications_firefighter
stop_RBRi?
stop
s1'==0
stop_s[3]?
L2
s1'==0
start_DSSCC?
10
stop_RBRic?
L1
10
L3
fire_alarm=RBR_oc.rule.notifications_firefighter

Fig. 12: The monitor automaton for requirement $R1_{\mathrm{CAMI}}$.

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 [6] for automated modeling, and provides some preliminary architecture-level analysis. It also allows us to model the behavior of the architecture components via behavior annex and encode the probabilities of failures of various components, via the error annex. However, AADL also has its limitations of expressing complex behaviors of algorithms such as CBR, which we have omitted in this work.

There are two analysis approaches presented in this paper: (1) using exhaustive model checking (2) using stochastic model checking, both automated automated via a commercial tools UPPAAL and UPPAAL SMC. The analysis approaches are chosen based on the system complexity. If the architecture model is scalable with exhaustive model-checjing, then it can be applied. Although the exhautive verification resulst are accurate, one cannot take into account the probabilistic behaviour of our systems. In case of complex models that needs to be analyzed for stochastic behaviours, the user can opt for simulation-based approaches, although it does not yield 100% accuracy. The verification results shown in this ppaper are specific to our architecture models defined, however one can use the approach to verify any set of requirements for various architecture types defined by the generic architectural model defined in this work. In case of exhaustive model-checking, the resulst are derived assuming that all components are operational such that we devoid the system of its probabilistic failure behaviour. Also, for the case of statiscal model checking, it is worth mentioning that the results are derived assuming high reliability of individual architecture components and considering specific values for the periods and execution times. However, taking into account the wide variety of available sensors and other components, we can easily adapt the values to account for requirements of any specific architecture.

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

8 Related Work

In recent years, there has been a lot of work in the area of AAL due to the need of supporting an increased elderly population [37]. Moreover, many functionalities that need to be tackled by AAL solutions are of a safety-critical nature, e.g., health emergencies like cardiac arrest, fall of the elderly, and home emergencies like fire at home, etc. [43], therefore work on their modeling and analysis is fully justified.

A study on existing AAL architectures shows that there are certain architecture types that address the construction of integrative AAL applications, some of the common ones being : Multi-Agent System (MAS) [21,30,41], Cloud-based [12,26] and Internet-of-Things (IoT) centric [23].

- **Agent-based architectures:** These are the most commonly used architectures for AAL applications owing to its flexibility, autonomy, adaptability, better response and service continuity due to its distributed nature . Some examples of health care frameworks that relies on a distributed agent architecture are [45], [21]. However, the agent based architectures also have some drawbacks (i) Restricted communication protocols for agent communication and the delay overhead in taking a collective decision and (ii) maintaining the consistency of the framework .
- **Cloud-based AAL solutions:** AAL solutions that leverage the potential of cloud computing for context modeling, intelligent decision making and use it as a data storeAlthough cloud based solutions are scalable, cost-effective, reusable, adaptable, and extendable, the sole processing with cloud cannot guarantee strict hard real-time properties and the system fails completely in the absence of Internet.
- **IoT architectures:** IoT technology is now getting widely getting utilized in the filed of AAL owing to its technological advancements. The IoT concept of communication between smart objects and people and people are widely exploited in the field of AAL, thereby providing connectivity, context-awareness and adaptivity. . There are also approaches to integrate the autonomous behavior of agent-based systems with IoT technology [27,36]. Although AAL systems based on IoT offer high flexibility, adaptability, the system depends only on the availability of the Internet for operation; which can lead to a complete failure of such systems in places where Internet connectivity is meager. Our architecture follows the design paradigms of Cloud-based AAL solutions, where the cloud is utilized for intelligent, context-aware decision making and as a data store, and is also augmented with local processing schemes to guarantee real-time properties. In many situations, cloud services cannot guarantee hard-real time properties and hence we adopted a local processing scheme as well in our model, and the cloud is a back-up which activates only when the primary has failed.

The formal assurance of AAL systems has been the focus of some related research in the recent years. Parente et al. provide a list of various formal methods that can be used for AAL systems [42]. In another interesting work, Rodrigues et al. [44] perform a dependability analysis of AAL architectures using UML and PRISM. Other interesting research work uses temporal reasoning [15,39] and Markov Decision Processes to formally verify the reliability of AAL systems [38]. Although these approaches target the formal analysis of AAL systems, most of the above work addresses only simple scenarios and are not used to analyze complex behaviors resulting from integrating critical AAL functions (e.g. fire and fall), as well as their decision making. In addition, these approaches do not aim to develop an overall framework for the verification of AAL systems, starting from an integrated architectural design, their design specifications, followed by a verification strategy, as proposed in this paper.

The use of Architecture Description Languages (ADL) to specify AAL designs has not been exercised previously, yet this is common when designing automotive or automation systems. There have also been approaches to formally verify AADL designs in other domains. The transformation approach from AADL to TA or variants has been already addressed by related work [17,29,31]. Although these approaches are automated verification techniques, there is a lack of focus on abstract components/patterns with stochastic properties. In addition, these approaches also suffer from state-space explosion, therefore they might not scale well to complex AAL designs. Nevertheless, there is interesting research that deals with stochastic properties and statistical model checking for the analysis of extended AADL models. One such example is in the work of Bruintjes et al. [18], where the authors have used SMC approach for timed reachability analysis of extended AADL designs. Although our approach also focuses on linear systems, it is different from the mentioned work in the fact that we focus on abstract components, and also introduce BA modeling for capturing the functional behavior of our modules, specifically for modeling the behavior of intelligent DSS. In their work, Bruintjes et al. use the SLIM Language, which is strongly based on AADL and is specific to avionics and automotive industry, including the error behavior and modes. However, we use the AADL core language with its standardized annex sets (EA and BA) for the architecture specification, thereby enabling us to represent the functional and error behaviour with the architecture model. The abstract component based modeling also brings exensiblity and reusability to our approach. Moreover, the authors only consider the event occurrences or delay variations using uniform or exponential distributions, wheras by employing our user-defined properties, we can also specify other distributions. Furthermore, the approach of Bruintjes et al. only deals with evaluation of time-bounded queries, however we also evaluate properties like reliability, data consistency, etc., along with timeliness. Another interesting work [16], possibly carried out in parallel with our work, employs statistical model checking using UPPAAL SMC to evaluate the performance of nonlinear hybrid models with uncertainty modeled in extended AADL. Although the approach is not specific to the AAL domain, it is promising to specify complex CPS systems considering uncertainties from physical environment. Unlike our model, the authors use Priced Timed Automata (PTA) models. In comparison, our approach considers only linear models that evolve continuously (yet the analysis is carried out in discrete time due to sampling of continuous data). In brief, the two approaches resemble, yet our approach is all contained in the core language of AADL (as different from the mentioned work where the authors resort to other annexes integrated in OSATE), is tailored to systems that contain AI components, and assumes the random failure of various components, which is not considered in the related work.

9 Conclusions and Future Work

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

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

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

Appendix A: AADL Model of RBR

```
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
subcomponents
AAL_event: data System_Data_model::events;
DA: data System_Data_model: User_activity;
u_profile: data System_Data_model:User_profile;
fuzzy_out1:data System_Data_model::fuzzified_data_health;
fuzzy_out2: data system_Data_model::fuzzified_data_camera;
annex EMV2{**
use types error_model;
use behavior error_model::simple;
error propagations
input: in propagation{NoValue};
output: out propagation{ Novalue};
flows
ef0: error path input{NoValue}->output{NoValue};
component error behavior
events
Reset: recover event;
TF: error event;
PF: error event;
err_p: error event;
transitions
t0: Operational-[TF]->Failed_transient;
t1: Failed_transient -[Reset]->Waiting with 0.8,
Failed_Permanent with 0.2;
t2: Operational -[PF]->Failed_Permanent;
t3: Operational-[err_p]->Failed_p;
t4: Failed_p -[input]->Operational;
end component;
properties
EMV2::DurationDistribution => [Duration => 1ms..2ms;
Distribution =>Fixed;] applies to reset;
EMV2::OccurrenceDistribution =>[ProbabilityValue => 0.2;
Distribution => Fixed;] applies to Failure_Transient;
EMV2::OccurrenceDistribution =>[ProbabilityValue => 0.1;
Distribution => Fixed;] applies to Failure_Permanent;
**};
annex behavior_specification {**
states
Waiting: initial complete final state;
Operational: state;
transitions
Waiting -[on dispatch input]->Operational {if
(AAL_event="fire"){output:="notification_firefighter fire"}
 elsif ( fuzzy_out1 = "Pulse_high" and DA!="exercising" and
 u_profile="cardiac_patient")
 {output := "notification_caregiver_highpulse"}
elsif (AAL_event = "fall" or fuzzy_out2 = "Fall_high")
{output := "notification_caregiver fall"}
elsif( fuzzy_out1 = "pulse-abnormal_low" )
{output:= "notification_caregiver"}
elsif(AAL_event = "fall" and fuzzy_out2= "Fall_high" and
AAL_event="fire" and fuzzy_out1= "pulse-abnormal_low")
{output:= "notification_caregiver fall,fire, pulse_low and
notification_firefighter fall, fire, pulse-abnormal-low"}
end if};
**};
end RBR.impl;
```

Acknowledgement

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

References

1. CAMI Gateway. https://eclexys.com/wp-content/uploads/2019/01/Exys9200-SNG-Brochure.pdf, accessed: 2019-03-16
2. Fibaro motion sensor. https://manuals.fibaro.com/content/manuals/en/FGMS-001/FGMS-001-EN-T-v2.0.pdf, accessed: 2019-03-16
3. Fitbit. https://www.fitbit.com/se/home, accessed: 2019-03-16
4. Linkwatch. https://www.linkwatch.se , accessed: 2018-01-15
5. Opentele. https://www.opentelehealth.com , accessed: 2018-01-15
6. OSATE—Open Source AADL Test Environment. http://osate.github.io/, accessed: 2018-05-15
7. Pepper robot. https://www.softbankrobotics.com/emea/en/pepper, accessed: 2019-03-16
8. Rabbit mq message broker. https://www.rabbitmq.com, accessed: 2019-03-16
9. Tiago robotic platform. http://tiago.pal-robotics.com, accessed: 2019-03-16
10. Ua651 bp sensor. http://www.andmedical.com.au/products-service/value-ua-651, accessed: 2019-03-16
11. Vibby fall detection sensors. http://www.vitalbase.co.uk, accessed: 2019-03-16
12. Ahmed, M.U., Björkman, M., Lindén, M.: A generic system-level framework for self-serve health monitoring system through internet of things (iot). Studies in health technology and informatics 211, 305–307 (2015)
13. Alur, R., Courcoubetis, C., Dill, D.: Model-checking for real-time systems. In: Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e. pp. 414–425. IEEE (1990)
14. Alur, R., Courcoubetis, C., Dill, D.: Model-checking in dense real-time. Information and computation 104(1), 2–34 (1993)
15. 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. pp. 778–782. IOS Press (2004)
16. 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 Transactions on Computer-Aided Design of Integrated Circuits and Systems 36(12), 1989–2002 (2017)
17. 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, pp. 53–79. Springer (2017)
18. Bruintjes, H., Katoen, J.P., Lesens, D.: A statistical approach for timed reachability in AADL models. In: Dependable Systems and Networks (DSN), 45th Annual IEEE/IFIP International Conference on. pp. 81–88. IEEE (2015)
19. Bulychev, P.E., David, A., Larsen, K.G., Legay, A., Li, G., Poulsen, D.B.: Rewrite-based statistical model checking of wmtl. RV 7687, 260–275 (2012)
20. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B.: Uppaal smc tutorial. International Journal on Software Tools for Technology Transfer 17(4), 397–415 (2015)

21. De Paz, J., Rodríguez, S., Bajo, J., Corchado, J., Corchado, E.: Ovacare: A multi-agent system for assistance and health care. Knowledge-Based and Intelligent Information and Engineering Systems pp. 318–327 (2010)
22. Delange, J., Feiler, P.: Architecture fault modeling with the aadl error-model annex. In: Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on. pp. 361–368. IEEE (2014)
23. Dohr, A., Modre-Osprian, R., Drobics, M., Hayn, D., Schreier, G.: The internet of things for ambient assisted living. ITNG 10, 804–809 (2010)
24. Feiler, P.H., Gluch, D.P.: Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language. Addison-Wesley (2012)
25. 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, pp. 3–15. Springer (2005)
26. Forkan, A., Khalil, I., Tari, Z.: Cocamaal: A cloud-oriented context-aware middleware in ambient assisted living. Future Generation Computer Systems 35, 114–127 (2014)
27. Fortino, G., Guerrieri, A., Russo, W.: Agent-oriented smart objects development. In: Computer Supported Cooperative Work in Design (CSCWD), 2012 IEEE 16th International Conference on. pp. 907–912. IEEE (2012)
28. Frana, R., Bodeveix, J.P., Filali, M., Rolland, J.F.: The AADL behaviour annex–experiments and roadmap. In: Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on. pp. 377–382. IEEE (2007)
29. Hamdane, M.E., Chaoui, A., Strecker, M.: From AADL to timed automaton-A verification approach. International Journal of Software Engineering and Its Applications 7(4) (2013)
30. Isern, D., Sánchez, D., Moreno, A.: Agents applied in health care: A review. International journal of medical informatics 79(3), 145–166 (2010)
31. Johnsen, A., Lundqvist, K., Pettersson, P., Jaradat, O.: Automated verification of AADL-specifications using UPPAAL. In: High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on. pp. 130–138. IEEE (2012)
32. Kunnappilly, A., Seceleanu, C., Lindén, M.: Do we need an integrated framework for ambient assisted living? In: Ubiquitous Computing and Ambient Intelligence: 10th International Conference, UCAmI 2016, San Bartolomé de Tirajana, Gran Canaria, Spain, November 29–December 2, 2016, Part II 10. pp. 52–63. Springer (2016)
33. Kunnappilly, A., Sorici, A., Awada, I.A., Mocanu, I., Seceleanu, C., Florea, A.M.: A Novel Integrated Architecture for Ambient Assisted Living Systems. In: Computer Software and Applications Conference (COMPSAC), 2017 IEEE 41st Annual. vol. 1, pp. 465–472. IEEE (2017)
34. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. International journal on software tools for technology transfer 1(1-2), 134–152 (1997)
35. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: International conference on runtime verification. pp. 122–135. Springer (2010)
36. Leong, P., Lu, L.: Multiagent web for the internet of things. In: Information Science and Applications (ICISA), 2014 International Conference on. pp. 1–4. IEEE (2014)
37. Li, R., Lu, B., McDonald-Maier, K.D.: Cognitive assisted living ambient system: A survey. Digital Communications and Networks 1(4), 229–252 (2015)
38. Liu, Y., Gui, L., Liu, Y.: Mdp-based reliability analysis of an ambient assisted living system. In: International Symposium on Formal Methods. pp. 688–702. Springer (2014)

39. 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 Transactions on Human-Machine Systems 43(6), 509–521 (2013)
40. Medjahed, H., Istrate, D., Boudy, J., Dorizzi, B.: Human activities of daily living recognition using fuzzy logic for elderly home monitoring. In: Fuzzy Systems, 2009. FUZZ-IEEE 2009. IEEE International Conference on. pp. 2001–2006. IEEE (2009)
41. Nealon, J., Moreno, A.: Agent-based applications in health care. Applications of software agent technology in the health care domain pp. 3–18 (2003)
42. Parente, G., Nugent, C.D., Hong, X., Donnelly, M.P., Chen, L., Vicario, E.: Formal modeling techniques for ambient assisted living. Ageing International 36(2), 192–216 (2011)
43. Rashidi, P., Mihailidis, A.: A survey on ambient-assisted living tools for older adults. IEEE journal of biomedical and health informatics 17(3), 579–590 (2013)
44. Rodrigues, G.N., Alves, V., Silveira, R., Laranjeira, L.A.: Dependability analysis in the ambient assisted living domain: An exploratory case study. Journal of Systems and Software 85(1), 112–131 (2012)
45. Tapia, D.I., Rodrıguez, S., Corchado, J.M.: A distributed ambient intelligence based multi-agent system for alzheimer health care. In: Pervasive Computing, pp. 181–199. Springer (2009)
46. Zhou, F., Jiao, J.R., Chen, S., Zhang, D.: A case-driven ambient intelligence system for elderly in-home assistance applications. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 41(2), 179–189 (2011)