Assuring Intelligent Ambient Assisted Living Solutions by Statistical Model Checking

Ashalatha Kunnappilly, Raluca Marinescu, and Cristina Seceleanu

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

Abstract. A modern way of enhancing elderly people's quality of life is by employing various Ambient Assisted Living solutions that facilitate an independent and safe living for their users. This is achieved by integrating computerized functions such as health and home monitoring, fall detection, reminders, etc. Such systems are safety critical, therefore ensuring at design time that they operate correctly, but also in a timely and robust manner is important. Most of the solutions are not analyzed formally at design time, especially if such Ambient Assisted Living functions are integrated within the same design. To address this concern, we propose a framework that relies on an abstract component-based description of the system's architecture in the Architecture Analysis and Design Language. To ensure scalability of analysis, we transform the AADL models into a network of stochastic timed automata amenable to statistical analysis of various quality-of-service attributes. The architecture that we analyze is developed as part of the project CAMI, co-financed by the European Commission, and consists of a variety of health and home sensors, a data collector, local and cloud processing, as well as an artificial-intelligence-based decision support system. Our contribution paves the way towards achieving design-time assured integrated Ambient Assisted Living solutions, which in turn could reduce verification effort at later stages.

1 Introduction

The elderly segment of the population often face cognitive decline, chronic age-related diseases, limitations in mobility, vision, and hearing. In this context, Ambient Assisted Living (AAL) solutions can enhance the elderly's quality of life by providing integrated computerized functions aimed to help the users in their independent and safe daily living. Modern AAL systems are equipped with a wide variety of integrated features, like health monitoring, home monitoring, fall detection, robotic platform support, and support for communication with caregivers [20]. AAL systems are complex safety-critical systems, operating in a dynamic and unpredictable environment (e.g. due to involvement of humans in the loop, like elderly users, caregivers), hence it is desirable that these systems are analyzed to ensure that they meet their functional and extra-functional requirements. As for any other computerized system, formal techniques such as model checking can also be applied to AAL; however, the task is not trivial due to the complexity of models that include artificial intelligence (AI) techniques, redundant components for fault tolerance, etc.

In this paper, we address this challenge by formally analyzing architectural specifications of integrated AAL solutions. Although here we focus on a particular system, the method is general and can be applied to similar AAL solutions [16]. The AAL system that we model and analyze is developed as part of the project CAMI [1], co-financed by the European Commission under the Ambient Assisted Living Joint Program of H2020. The system consists of various sensors, actuators, user interfaces, cloud services used to store and process data, and also an intelligent decision support system (DSS) that we design and analyze in this work. Our DSS uses a combination of AI reasoning techniques, such as fuzzy logic, rule-based reasoning (RBR), and case-based reasoning (CBR) for efficient decision making.

Our contribution starts with specifying the AAL system architecture in the Architecture Analysis and Design Language (AADL) [11] (see Section 3.1). AADL is chosen due to the rich set of modeling constructs and its suitability in specifying real-time systems. In this work, we assume fault-tolerant AAL systems, which we model as a set of interacting *abstract* components in AADL. To specify each component's functional behavior and error behavior, we use the standardized annex sets of AADL, Behavior Annex (BA), and Error Annex (EA), respectively.

Although existing AADL IDE tools like OSATE [1] support initial architecture analysis with respect to latency (timing properties), schedulability, etc., the worst-case timing assumptions are often adopted without considering the component behaviors, which often leads to pessimistic estimations [4]. Thus, there is a need to adopt formal analysis methods like model checking to analyze the functional correctness of an architecture, or the system's timeliness or reliability, within the architectural framework itself. To carry out such analysis and to tackle the latter's scalability, we resort to statistical analysis rather than exhaustive model checking to ensure the system's correctness. Hence, we give semantics to the AADL components in terms of stochastic timed automata (STA) that can be statistically model checked with UPPAAL SMC [8] (See Section 3.2). Since our AADL model considers probabilities of failure and recovery of various components, a straightforward encoding would be within a probabilistic framework such as that of PRISM [14], for exhaustive verification. However, the latter does not scale to the size of our model so we resort to an SMC encoding, based on UPPAAL SMC [8]. Although the analysis results are not exact, simulation-based methods are sometimes the only choice for reasoning of complex Cyber Physiscal Systems [4,19].

In brief, the contributions of this work are as follows: i) an abstract component-based AADL architectural model of the integrated AAL system CAMI (Section 4), ii) an AI-based DSS and its abstract component-based specification (Section 2), iii) a formal definition of the proposed AADL components in terms of STA (Section 5), and iv) the formal analysis of the STA model against various func-

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

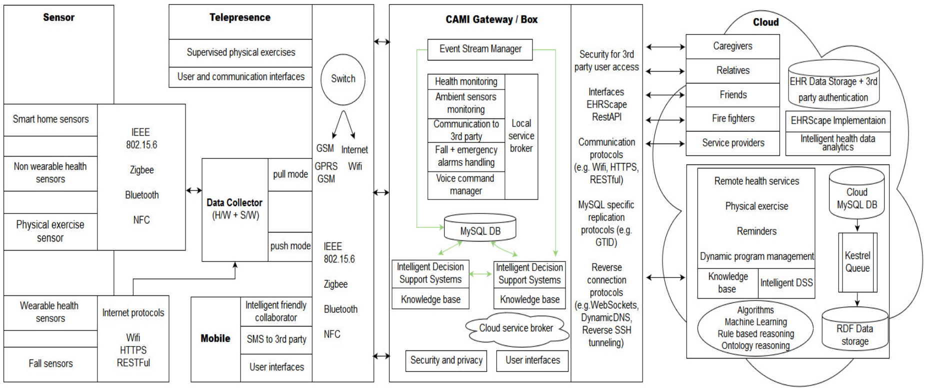


Fig. 1: The CAMI AAL System Architecture [18]

tional requirements and quality-of-service (QoS) properties with UPPAAL SMC (Section 6). We compare to related work in Section 7, and conclude the paper in Section 8.

2 Use case: CAMI AAL Architecture

In this section, we describe our use case, the CAMI AAL system architecture [18]. The system architecture is shown in Fig.1, and follows the structure of many commercial AAL systems with various sensors, data collector, DSS, security and privacy, database systems, user interfaces (UI), local and cloud computing support. The major components of CAMI are briefly described below:

– Sensor unit: Comprises various sensors ranging from health monitoring to home monitoring, physical exercise monitoring, and fall detection.
– Data Collector Unit: Collects data from the sensors and assigns labels and criticality-based priorities for handling events.
– Local processor (CAMI Gateway): Processes the data from the Data Collector Unit. It has two major sub-components:
  - Message Queue (MQ): Queues up the data output from the Data Collector and sorts it according to its priority such that the head of the queue is the sensor data/event with the highest priority. In Fig.1, the MQ is the Event Stream Manager.
  - Decision Support System (DSS): The brain of the architecture where the data from the MQ gets processed. To handle the complexity associated with the integration of multiple functionalities and to process the interdependencies among simultaneously-occurring events in the AAL environment, we design a novel **intelligent context-aware DSS** system for CAMI that utilizes various AI reasoning schemes to decide on further actions to be taken (e.g., raise an alarm and inform the caregiver, issue a reminder etc.). The DSS architecture is detailed in Section 2.1.

– Cloud processor: Is employed for data storage and processing. A redundant copy of the DSS is maintained in the cloud to avoid the single point of failure of the architecture. In this paper, we have considered only one redundant DSS copy in the cloud due to cost and maintenance efficiency.

2.1 CAMI's Decision Support System

The DSS architecture that we propose is shown in Fig. 2. Our architecture is inspired by the work of Zhou et al. [27], where the authors have proposed a context-aware, CBR-based ambient intelligence system for AAL applications. In comparison, our DSS combines multiple AI reasoning techniques in an effective way. We employ context modeling (CM) with fuzzy reasoning, RBR in addition to CBR, which help to deal efficiently with multiple AAL scenarios. Context modeling is performed by identifying the context space based on: (i) the personal profile of the user, (ii) the activity of daily living (DA) performed by the user, (iii) spatio-temporal properties, (iv) environment and (v) health parameters. The parameters of context space are either profiled, sensed, or predicted. Fuzzy reasoning is used for detecting DA [23], and also for determining sensor-data deviations. To take decisions in various situations, we employ RBR first, next CBR, that is, upon a context-change, 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 of the particular case. The Knowledge Base (KB) stores the context, rules, and cases. For detailed description of the internal structure of DSS, the reader can refer to the technical report [16].

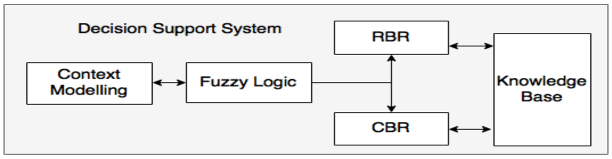


Fig. 2: The DSS architecture

2.2 Use Case Scenarios and System Requirements

The CAMI system assists the elderly user in a variety of health, home-related, and social inclusion functionalities. In this paper, we focus on one of the critical scenarios where CAMI comes into assistance of an elderly user, Jim, who falls frequently and suffers from chronic cardiac disease and slight memory loss.

Scenario 1: Jim has sudden pulse variations, detected by the pulse monitoring sensor of CAMI, which are critical for cardiac patients. If the pulse is low, CAMI alerts the caregiver of a low pulse. If the pulse is high, CAMI checks whether 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 : 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 events [17]. 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 a further diagnosis. This scenario is safety critical and has to be processed in real time.

We present the functional requirement that is derived from the above scenarios and also the QoS requirements that CAMI should satisfy:
R1: If there is a pulse data deviation indicating high pulse, the DA is "not exercising", and the user has a cardiac disease history, then the DSS sends a notification to the caregiver, within 20 s. This relates to Scenario 1.
R2: If fire and fall are detected simultaneously, then the DSS should detect the presence of the simultaneous events and send notifications to both firefighters and caregiver, within 20 s. This relates to Scenario 2.
R3: 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.
R4: 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 that the CAMI architecture meets the above requirements. We achieve this by first specifying a simplified version of CAMI in AADL, and then by semantically encoding the specification into a network of STA (NSTA) that we statistically model check with UPPAAL SMC.

3 Preliminaries

In this section, we briefly overview AADL, STA and UPPAAL SMC.

3.1 Architecture Analysis and Design Language

AADL [11] 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*, *Execution Platform* and *System* are used to represent the run-time architecture of the system, however a more generalized representation is possible by specifying it as *abstract*.

A component in AADL can be defined by its *type* and *implementation*: the first defines the interface of the component and its externally observable attributes, whereas the second defines its internal structure. AADL allows possible component interactions via *ports/features*, *shared data*, *subprograms*, and *parameter connections*. In AADL, the input/output ports can be defined as: *event ports*, *data ports*, and *event-data ports*. Based on the component interactions, explicit *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*. We also employ various user-defined properties for representing the probabilistic distribution of an aperiodic event and the rate at which a component recovers from the failure.

The AADL core language can be extended via annex sublanguages and user-defined properties. In this work, we employ the standardized annexes of AADL for describing the functional and error behavior of a component, namely the Behavior Annex (BA) [12] and the Error Annex (EA) [9] respectively, which model behaviors as transition systems. 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, as well as the probability distributions associated with the error states and events.

In this paper, we focus on abstract components that allow us to defer from the run-time architecture of the system. 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* [10]. 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.

3.2 Stochastic Timed Automata and UPPAAL SMC

A timed automaton as used in the model checker UPPAAL is a formal notation used in describing real-time systems [2], 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, $c?$ (or $c!$) is a synchronization action a is an update action, and r is the clock reset set, that is, the clocks that are set to 0 by the edge. A location can be marked as *urgent* (marked with a U) or *committed* (marked with a C) indicating that time cannot progress in such locations. The latter is more restrictive, indicating that the next edge to be traversed 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 $u' \vDash I(l')$.

UPPAAL SMC [8] provides statistical model checking for stochastic timed automata (STA). The stochastic interpretation refines the TA with: (i) probabilistic choices between multiple enabled transition, where the output *probability* function γ may be defined by the user, and (ii) probability distributions for non-deterministic time delays, where the *delay density function* μ is a uniform distribution for time-bounded delays or an exponential distribution with user-defined rates for cases of unbounded delays. 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 the invariant in l. With E_{l} we denote the disjunction of guards g such that $l \xrightarrow{g,o,\text{-}}$ - $\in E$ 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 as 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 set to an exponential distribution with a rate $P(l)$, where $P : L \rightarrow \mathbb{R}_{\geq 0}$ is an additional distribution rate specified for the automaton. The output probability function γ_{s} for every state $s = (l, v) \in S$ is the uniform distribution over the set $\{o : (l, g, o, \text{-}, \text{-}) \in E \wedge v \vDash g\}$.

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. UPPAAL SMC uses an extension of weighted metric temporal logic (WMTL) [7] to evaluate a property $Pr(*_{x \leq C} \phi)$, 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.

4 AADL Model of CAMI

In this section, we present the modeling framework used for representing the CAMI AAL architecture. We use a generic representation of the components by specifying them as *abstract*, which can be easily extended to specific run-time architecture models.

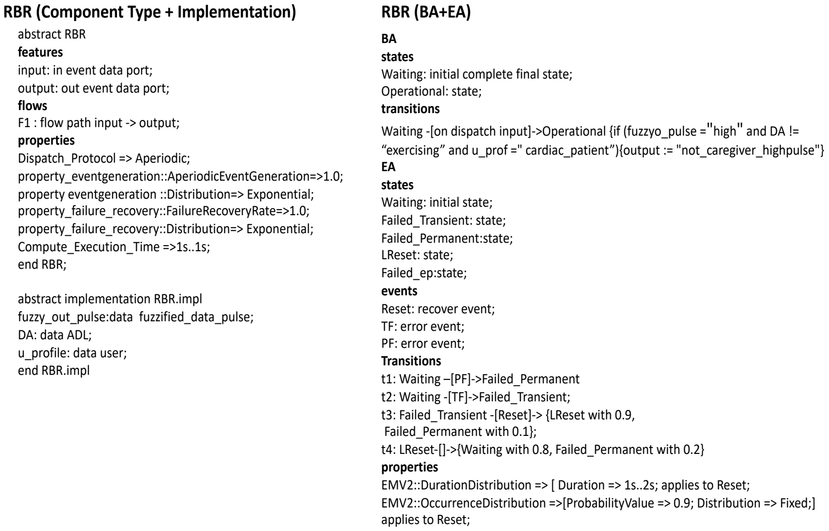
For developing the abstract model, we classify the AADL components as:

- **Atomic Components** (AC): components that do not have hierarchy in terms of sub-components with interfaces, but might contain sub-components without interfaces.
- **Composite Components** (CC): hierarchical components that contain sub-components with or without interfaces.

For instance, "data" is a sub-component in AADL without interface and it can be part of an AC or CC hierarchy.

In the CAMI architecture, the Sensors, UI, and Data Collector are modeled as AC, whereas the Local Controller, and the Cloud Controller are modeled as CC. The DSS is modeled as a CC with sub-components like CM, CBR and RBR. The fuzzy system is modeled as one of the sub-components of CM. Among all the elements involved in CM, activity recognition (detecting the DA of the user) is highly complex and requires analysis of multiple sensor parameters, so in our current model we have abstracted the module by associating a DA with the user profile in the context model. Similarly, we abstract away the algorithms of the CBR reasoning and only show how RBR outputs can successfully build the case base of the CBR module. We also model a redundant copy of the DSS component in the cloud processor.

An example of an AC in the CAMI architecture is the RBR component of the DSS. In this paper, we illustrate the RBR for **R1** (Scenario 1), described in Section 2.1. The RBR component type, implementation, BA, and EA are shown in Fig. 3. The component type definition specifies its name, category (i.e., "abstract") and interfaces. The RBR component type describes that it gets activated aperiodically according to a probabilistic distribution, has an execution time of 1 s, a failure recovery rate defined by the distribution, and illustrates the data flows between the respective input and output ports. The implementation definition of RBR defines the data sub-components like the fuzzy data output, personal information and daily activity of the user, which form the **context-space** of Scenario 1.



RBR (Component Type + Implementation)

```
abstract RBR
features
input: in event data port;
output: out event data port;
flows
F1 : flow path input -> output;
properties
Dispatch_Protocol => Aperiodic;
property_eventgeneration::AperiodicEventGeneration=>1.0;
property eventgeneration ::Distribution=> Exponential;
property_failure_recovery::FailureRecoveryRate=>1.0;
property_failure_recovery::Distribution=> Exponential;
Compute_Execution_Time =>1s..1s;
end RBR;

abstract implementation RBR.impl
fuzzy_out_pulse:data  fuzzified_data_pulse;
DA: data ADL;
u_profile: data user;
end RBR.impl
```

RBR (BA+EA)

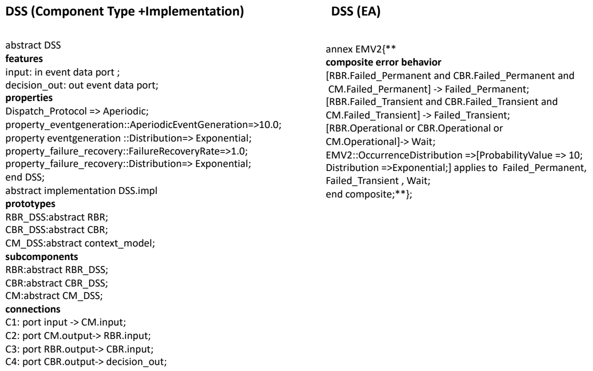
```
BA
states
Waiting: initial complete final state;
Operational: state;
transitions
Waiting -[on dispatch input]->Operational {if (fuzzyo_pulse ="high" and DA !=
“exercising” and u_prof =" cardiac_patient”){output := "not_caregiver_highpulse"}
EA
states
Waiting: initial state;
Failed_Transient: state;
Failed_Permanent:state;
LReset: state;
Failed_ep:state;
events
Reset: recover event;
TF: error event;
PF: error event;
Transitions
t1: Waiting –[PF]->Failed_Permanent
t2: Waiting -[TF]->Failed_Transient;
t3: Failed_Transient -[Reset]-> {LReset with 0.9,
 Failed_Permanent with 0.1};
t4: LReset-[]->{Waiting with 0.8, Failed_Permanent with 0.2}
properties
EMV2::DurationDistribution => [ Duration => 1s..2s; applies to Reset;
EMV2::OccurrenceDistribution =>[ProbabilityValue => 0.9; Distribution => Fixed;]
applies to Reset;
```

Fig. 3: An excerpt of the RBR component model

The BA has two states, *Waiting* and *Operational. Waiting* represents the initial state where the component waits for an input from the pulse sensor, and *Operational* is the state to which a component switches upon receiving the input (if it has not failed). In the *Operational* state, the system monitors the **fuzzy logic** output to identify any pulse variations. The fuzzy reasoning is not shown in Fig.4 due to space constraints, 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 read 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 person's activity of daily living and disease history. If the activity is "not exercising" and the user has a cardiac disease history, a notification alert is raised and sent to the caregiver. The information is encoded as a rule in the BA depicted in Fig. 3. Upon triggering a particular rule, the RBR output is stored in the DB as a case input for CBR, where the case-id is represented by DA, case features are the context space and the case solution is the RBR output. The RBR output is also synchronized with Cloud DSS such that data consistency is maintained.

The 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. In the



DSS (Component Type +Implementation)

```
abstract DSS
features
input: in event data port ;
decision_out: out event data port;
properties
Dispatch_Protocol => Aperiodic;
property_eventgeneration::AperiodicEventGeneration=>10.0;
property eventgeneration ::Distribution=> Exponential;
property_failure_recovery::FailureRecoveryRate=>1.0;
property_failure_recovery::Distribution=> Exponential;
end DSS;
abstract implementation DSS.impl
prototypes
RBR_DSS:abstract RBR;
CBR_DSS:abstract CBR;
CM_DSS:abstract context_model;
subcomponents
RBR:abstract RBR_DSS;
CBR:abstract CBR_DSS;
CM:abstract CM_DSS;
connections
C1: port input -> CM.input;
C2: port CM.output-> RBR.input;
C3: port RBR.output-> CBR.input;
C4: port CBR.output-> decision_out;
```

DSS (EA)

```
annex EMV2{**
composite error behavior
[RBR.Failed_Permanent and CBR.Failed_Permanent and
 CM.Failed_Permanent] -> Failed_Permanent;
[RBR.Failed_Transient and CBR.Failed_Transient and
CM.Failed_Transient] -> Failed_Transient;
[RBR.Operational or CBR.Operational or
CM.Operational]-> Wait;
EMV2::OccurrenceDistribution =>[ProbabilityValue => 10;
Distribution =>Exponential;] applies to  Failed_Permanent,
Failed_Transient , Wait;
end composite;**};
```

Fig. 4: An excerpt of the DSS component model

EA of Fig. 3, we show two of the states - *Waiting* and *Failed Transient*, plus their transitions based on a *TF* event (event that causes transient failures) and *reset* event. If a *TF* event occurs when the component starts, the latter moves to the *Failed Transient* state. 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. In Fig. 4, we present an excerpt of the DSS component, as an example of CC. The component type definition 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 shows the prototypes used to define sub-components and connections between them. The EA shows the composite error behavior of DSS and shows that the DSS moves to *Failed Transient* or *Failed Permanent*, if all 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.

The assumptions of CAMI's AADL model are: (i) all the system components have a reliability of 99.98%, (ii) each sensor has 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, without any loss of information during transmission.

In the following, we define the syntax of AAL-relevant AADL components as tuples, and their semantics in terms of stochastic timed automata.

5 Semantics of AAL-Relevant AADL Components

In this section, we introduce the tuple definition of an AADL component, for which we provide formal semantics as a network of STA.

An AADL component that we employ in this paper is defined by the following tuple:

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

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

The RBR component of DSS is an AC defined by its type, implementation, BA and EA (Fig. 4), as follows:

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

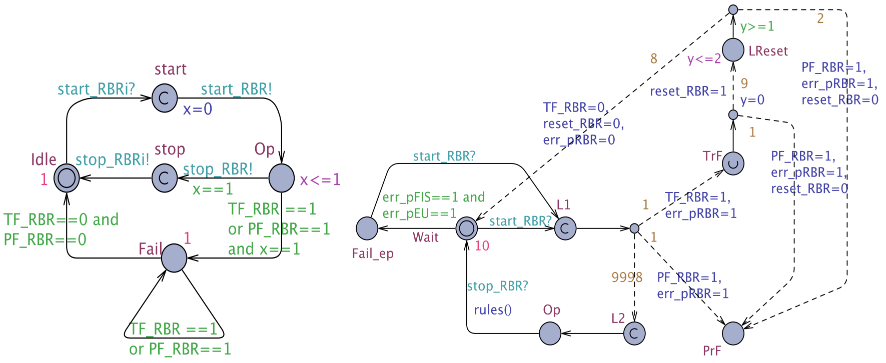
As a whole, the DSS in our CAMI architecture is a CC, and hence it is defined by its type, implementation and EA (no BA) as shown in Fig. 5. Formally, it can be represented as follows:

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

In the following we present the semantic encoding of RBR abd DSS, respectively, in terms of STA.

Definition 1 (Formal Encoding of RBR). *Let us assume an RBR component defined by Equation 4. We define the formal encoding of RBR as the following network of synchronized STA: $RBR_{NSTA} = RBR_{iSTA} || RBR_{aSTA}$, where RBR_{iSTA} is the "interface" STA of the RBR component and RBR_{aSTA} is the "annex" STA that encodes both the behavior and the error annex information.*

- ***RBR_{iSTA}*** *is defined as an STA [16] of the form: $\langle L, l_0, A, V, C, E, I, \mu, \gamma \rangle$, where:*
  - $L = \{Idle, Start, Op, Fail\}$, $l0 = Idle$;
  - $A = \{start\_RBRi?, start\_RBR!, stop\_RBR!, stop\_RBRi!\} \cup \{x = 0\}$, *where A comprises the set of synchronization channels associated with its input-output ports (start_RBR!, stop_RBR!), plus the synchronization channels to concord with DSS (start_RBRi?,stop_RBRi!) and the reset actions on clock x;*
  - $V = out\_port \cup in\_port \cup \{PF\_RBR, TF\_RBR\}$, *where out_port and in_port represent the set of output and input ports, respectively, and the boolean variables, PF_RBR, TF_RBR, represent the error events;*
  - $C = \{x\}$ *is the clock that models the execution-time of RBR ($T_e = 1$);*
  - $E = \{Idle \xrightarrow{start\_RBRi?} start, start \xrightarrow{start\_RBR!, x=0} Op,$ $Op \xrightarrow{TF\_RBR==1 \vee PF\_RBR==1 \wedge x==1} Fail, Op \xrightarrow{x==1, stop\_RBR!} stop,$ $stop \xrightarrow{stop\_RBRi!} Idle, Fail \xrightarrow{TF\_RBR==0 \wedge PF\_RBR==0} Idle,$ $Fail \xrightarrow{TF\_RBR==1 \wedge PF\_RBR==1} Fail\}$;
  - $I(Op) = x \leq 1$;
  - $P(Idle) = 1$, $P(Fail) = 1$, *given by γ, where $P(Idle) = 1$ represents the occurrence distribution of aperiodic events and $P(Fail) = 1$ represents the probability of leaving location Fail;*
- *The* ***RBR_{aSTA}*** *is created in a similar way with:*



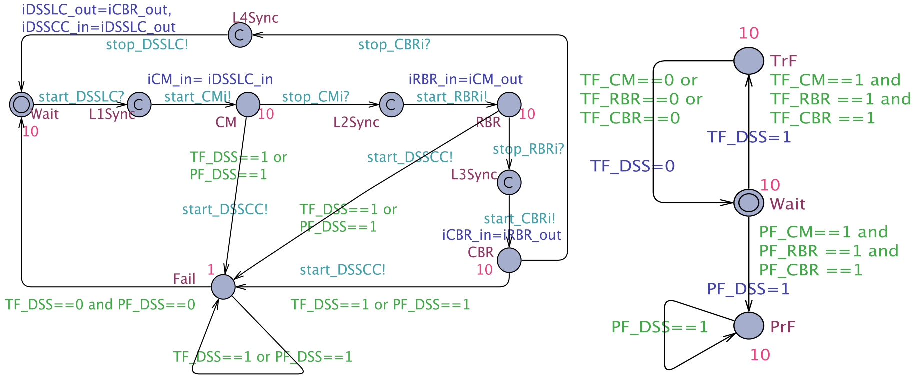
(a) RBR_{iSTA} (b) RBR_{aSTA}

Fig. 5: The NSTA for the RBR

- $L = \{Wait, Op, TrF, PrF, Fail\_ep, LReset, L1, L2\}, l0 = Wait$, *where* L *comprises the set of states in the EA and BA, respectively, plus additional committed locations* $(L1, L2)$ *that ensure that receiving is deterministic in UPPAAL SMC;*
- $A = \{start\_RBR?, stop\_RBR?\} \cup \{rules(), TF\_RBR = 0, TF\_RBR = 1, PF\_RBR = 1, reset\_RBR = 0, reset\_RBR = 1, err\_pRBR = 0, err\_p RBR = 1, err\_p = 1, y = 0\}$, *where* A *is composed of the actions defined in the BA and EA, plus the synchronizations channels to concord with* RBR_{iSTA} *(*$start\_RBR?, stop\_RBR?$*), and the reset of clock* y*;*
- $V = \{PF\_RBR, TF\_RBR, reset\_RBR, err\_pRBR, err_p\}$, *where* V *consists of the set of error events defined in the EA;*
- $C = \{y\}$ *models the duration of the "component reset";*
- $E = \{Wait \xrightarrow{start\_RBR?} L1, L1 \xrightarrow{TF\_RBR=1, err\_pRBR=1} TrF, L1 \xrightarrow{PF\_RBR=1, err\_pRBR=1} PrF, L1 \rightarrow L2, L2 \rightarrow Op, Op \xrightarrow{stop\_RBR?, rules()} Wait, TrF \xrightarrow{reset\_RBR=1, y=0} LReset,$ $TrF \xrightarrow{PF\_RBR=1, err\_pRBR=1, reset\_RBR=0} PrF,$ $LReset \xrightarrow{TF\_RBR=0, err\_pRBR=0, reset\_RBR=0} Wait,$ $LReset \xrightarrow{PF\_RBR=1, err\_pRBR=1, reset\_RBR=0} PrF,$ $Wait \xrightarrow{err\_p==1} Fail\_ep\}$, *where* E *consists of the transitions in EA, BA and those between* $L1$ *and* $L2$*;*
- $I(LReset) = y \leq 2$*;*
- $P(Wait) = 10$, *given by* γ*, that is the occurrence-distribution of* $Wait$*;*
- $L1 \xrightarrow{0.9998} L2$, $L1 \xrightarrow{0.001} TrF$, $L1 \xrightarrow{0.001} PrF$, *assigned by* μ*, where* μ *is the occurrence-distribution of error events.* □

Figure 5 depicts the NSTA for the RBR, as described by Definition 1.

Definition 2 (Formal Encoding of DSS). *The formal encoding of the DSS defined by the tuple in Equation 5 is also a network of two synchronized STA,*



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

Fig. 6: The NSTA for the DSS

$DSS_{NSTA} = DSS_{iSTA}||DSS_{aSTA}$, where DSS_{iSTA} is the "interface" STA of the DSS component, and DSS_{aSTA} is the "annex" STA that encodes the information from the error annex in AADL.

– ***DSS_{iSTA}*** *is defined as follows:*
  - *$L = \{Wait, CM, RBR, CBR, Fail, L1Sync, L2Sync, L3Sync, L4Sync\}$, $l0 = Wait$, where L comprises the the sub-componenets of the DSS (CM, RBR, CBR), plus additional locations to ensure synchronization (L1Sync, L2Sync, L3Sync, L4Sync), and location Fail to model the component failure.*
  - *$A = \{start\_DSSLC?, start\_CMi!, stop\_CMi?, start\_RBRi!, stop\_RBRi?, start\_CBRi!, stop\_CBRi?, stop\_DSSLC!, start\_DSSCC!\} \cup \{iCM\_in = iDSSLC\_in, iRBR\_in = iCM\_out, iCBR\_in = iRBR\_out, iDSSLC\_out = iCBR\_out, iDSSCC\_in = iDSSLC\_out\}$,where A consists of the synchronizations actions with DSS sub-components, and the assignments associated with the corresponding connections and flows;*
  - *$V = \{iDSSLC\_in, iCM\_in, iRBR\_in, iCBR\_in, iDSSCC\_in, iDSSLC\_out, iCM\_out, iRBR\_out, iCBR\_out, iDSSLC\_out, PF\_DSS, TF\_DSS\}$, where V is defined in the similar way as that of RBR_{iSTA};*
  - *$E = \{Wait \xrightarrow{start\_DSSLC?} L1Sync, L1Sync \xrightarrow{start\_CMi!, iCM\_in=iDSSLC\_in} CM, CM \xrightarrow{stop\_CMi?} L2Sync, L2Sync \xrightarrow{start\_RBRi!, iRBR\_in=iCM\_out} RBR, RBR \xrightarrow{stop\_RBRi?} L3Sync, L3Sync \xrightarrow{start\_CBRi!, iCBR\_in=iRBR\_out} CBR, CBR \xrightarrow{stop\_CBRi?} L4Sync, L4Sync \xrightarrow{stop\_DSSLC!, iDSSLC\_out=iCBR\_out, iDSSCC\_in=iDSSLC\_out} Wait, CM \xrightarrow{(TF\_DSS=1 \vee PF\_DSS=1), start\_DSSCC!} Fail, RBR \xrightarrow{(TF\_DSS=1 \vee PF\_DSS=1), start\_DSSCC!} Fail, CBR \xrightarrow{(TF\_DSS=1 \vee PF\_DSS=1), start\_DSSCC!} Fail, Fail \xrightarrow{(TF\_DSS==1 \vee PF\_DSS==1)} Fail, Fail \xrightarrow{(TF\_DSS==0 \wedge PF\_DSS==0)} Wait\}$,*

where E is defined with respect to the connections between the respective sub-components and their synchronizations;

- $P(Wait)$=10, $P(CM)$=10, $P(RBR)$=10, $P(CBR)$=10, $P(Fail)$=1, *defined by* γ.

– *The* $\boldsymbol{DSS_{aSTA}}$ *is defined in a similar way as* RBR_{aSTA}*, except that the encoding is done with the elements of the EA. (Note: There is no BA defined for CC)*

- $L = \{Wait, TrF, PrF\}$, $l0 = Wait$;
- $A = \{TF\_DSS = 0, TF\_DSS = 1, PF\_DSS = 1\}$;
- $V = \{TF\_DSS, TF\_CM, TF\_RBR, TF\_CBR, PF\_CM, PF\_RBR, PF\_CBR, PF\_DSS\}$;
- $E = \{Wait \xrightarrow{TF\_CM==1 \wedge TF\_RBR==1 \wedge TF\_CBR==1, TF\_DSS=1} TrF, Wait \xrightarrow{PF\_CM==1 \wedge PF\_RBR==1 \wedge PF\_CBR==1, PF\_DSS=1} PrF, PrF \xrightarrow{PF\_DSS==1} PrF, TrF \xrightarrow{TF\_CM==0 \vee TF\_RBR==0 \vee TF\_CBR==0, TF\_DSS=0} Wait\}$;
- $P(Wait) = 10, P(TrF) = 10, P(PrF) = 10$, *defined by* γ. □

Figure 6 depicts the NSTA for the DSS, as described by the Definition 2.

6 Statistical Analysis of CAMI with UPPAAL SMC

In this section, we check if the STA encoding of CAMI satisfies the requirements introduced in Section 2.2.

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

Table 1: Verification results

We verify the system requirements using UPPAAL SMC [8]. To check that our CAMI DSS meets its requirements, we employ a monitor STA that monitors the sensor values, the respective DSS output, and the corresponding clock. The verification results are tabulated in Table 1. The system satisfies all the functional requirements corresponding to Scenarios 1 and 2 (R1 and R2) with high probabilities (close to 1) and with high confidence. Requirements R3 and R4 are related the QoS attributes of the CAMI architecture. R3 checks the data consistency of Local DSS and Cloud DSS and requires that the RBR outputs of the local DSS get stored in the case-base of the cloud DSS. This requirement is satisfied with a high probability of [0.99975, 1] and high confidence of 0.998. Query R4 is related the fault-tolerance of CAMI. We see from Table 1 that the probability of cloud DSS to become activated (R4) is [0.01, 0.04]; this is because it gets activated only when the local DSS has failed and the failure probability of local DSS is between [0.01, 0.04] for a simulation over 1000 time units. However, if the local DSS has failed, we see that the probability of cloud DSS getting activated is very high [0.99975, 1] with a confidence of 0.998, which satisfies our requirement. Most of the requirements are verified with queries that contain terms of the form *A imply B*, therefore a pre-check of each corresponding “A”, being reachable is first carried out.

7 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 [20]. 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. [25], therefore work on their modeling and analysis is fully justified.

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 [24]. In another interesting work, Rodrigues et al. [26] perform a dependability analysis of AAL architectures using UML and PRISM. Other interesting research work uses temporal reasoning [3,22] and Markov Decision Processes to formally verify the reliability of AAL systems [21]. 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 [5,13,15]. 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. [6], 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 [4], 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.

8 Conclusions and Future Work

In this paper, we have presented a framework for the formal modeling and analysis of integrated AAL systems, instantiated on CAMI that includes a variety of sensors, data collector unit, intelligent decision support system, user interfaces, and local and cloud processing schemes.

As a first step, CAMI is represented as interacting abstract components in AADL, using a commercially available tool, OSATE [1]. To provide scalable formal analysis of the AAL system, we have semantically encoded the AADL CAMI model into a network of stochastic timed automata. The resulting formal model has been analyzed with UPPAAL SMC to ensure that the required functional behavior with timeliness, reliability and fault-tolerance is enforced with high probability and accuracy. The framework is intended to augment existing AAL solutions with formal analysis support and provide analysis prior to implementation. Such an analysis is beneficial for AAL systems, which are real-time, safety-critical systems, and require high levels of dependability.

Due to the heterogeneity of components available in the AAL domain, the component failure probabilities, periods and execution times are not chosen with respect to any specific category of components, nevertheless the results presented in the paper are promising as the components that have been proposed can be refined further. The verification results are specific to our CAMI architecture, however one can use the approach to verify any set of requirements for various architecture types defined by the generic architectural model documented in the extended technical report of this work [16]. 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 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 encoding.

Acknowledgement

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

References

1. OSATE—Open Source AADL Test Environment. http://osate.github.io/, accessed: 2018-05-15
2. Alur, R., Courcoubetis, C., Dill, D.: Model-checking in dense real-time. Information and computation 104(1), 2–34 (1993)
3. Augusto, J.C., Nugent, C.D.: The use of temporal reasoning and management of complex events in smart homes. In: Proceedings of the 16th European Conference on Artificial Intelligence. pp. 778–782. IOS Press (2004)
4. 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)
5. 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)
6. 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)
7. 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)

8. 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)
9. 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)
10. Feiler, P.H., Gluch, D.P.: Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language. Addison-Wesley (2012)
11. 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)
12. 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)
13. 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)
14. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 441–444. Springer (2006)
15. 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)
16. Kunnappilly, A., Marinescu, R., Seceleanu, C.: A Statistical Analysis Framework for Ambient Assisted Living Solutions. Tech. rep., http://www.es.mdh.se/publications/5125-
17. 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)
18. 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)
19. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: International conference on runtime verification. pp. 122–135. Springer (2010)
20. Li, R., Lu, B., McDonald-Maier, K.D.: Cognitive assisted living ambient system: A survey. Digital Communications and Networks 1(4), 229–252 (2015)
21. 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)
22. 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)
23. 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)

24. 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)
25. 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)
26. 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)
27. 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)