Architecture Modelling and Formal Analysis of Intelligent Multi-Agent Systems

Ashalatha Kunnappiilly, Simin Cai, Raluca Marinescu and Cristina Seceleanu
Mälardalen University, Västerås, Sweden
{first.last}@mdh.se

Keywords: Cyber-physical systems, Ambient Assisted Living, Multi-agent systems, Architecture Analsysis and Design Language, model checking, PRISM

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

1 Introduction

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

In order to guarantee the system's safety and achieve the desired quality of service (QoS), it is beneficial to ensure the correctness of the AAL system design, with respect to the real-time, fault-tolerant and probabilistic behaviors of agents, both individually and in cooperation. To achieve this, specification and rigorous analysis of such behaviors are necessary, which should go hand-in-hand with the specification and analysis of the entire architecture in which the agents are integrated. Existing techniques either do not support the integrated specification and analysis of architecture and agent behaviors, or lack reasoning capabilities of combined real-time, fault-tolerant and probabilistic behaviors that are essential to many AAL systems (Kunnappilly et al., 2018; Rodrigues et al., 2012).

In this paper, based on existing solutions (Tapia

et al., 2009), we propose a MAS architecture for AAL, comprising simple reflex agents based on if-then-else rules, and complex intelligent agents with self-learning based on Reinforcement Learning (RL) (Sutton et al., 1998). As our basis for specification, we choose a commonly-used architecture specification language, that is, the Architecture Analysis and Design Language (AADL) (Feiler et al., 2006). We use the original AADL constructs to specify the architecture including the agent components, their interfaces and communication. However, since the core AADL language lacks the ability to specify emergent agent behaviours, which may be both probabilistic and non deterministic, we propose an annex extension to the core AADL, referred to as **Agent Annex**. Unlike the existing Behaviour Annex specification of AADL (Dissaux et al., 2006), usually used for encoding component behavior, the Agent Annex allows one to describe the combined real-time, fault-tolerant and probabilistic behaviors. We formulate the new annex by extending the AADL meta model, and define its semantics as a stochastic transition system. To enable formal verification, we also provide formal semantics to the AADL architectural model, in terms of Stochastic Transition Systems (STS). We employ the state-of-the-art probabilistic model checker, PRISM (Kwiatkowska et al., 2002), to formally verify a set of crucial functional and quality-of-service properties of an illustrative AAL use case.

The rest of the paper is organized as follows. Section 2 overviews the basics of AADL, STS and PRISM. In Section 3, we describe our AAL system architecture based on MAS. We present the AADL modeling constructs and the Agent Annex extension in Section 4. Section 5 describes the formal encoding of the AADL model, and in Section 6, we present the verification results applying the PRISM model-checker on a representative AAL system. Related work is described in Section 7. Some discussion points are presented in Section 8 and conclusions and future work in Section 9.

2 Preliminaries

In this section, we give an overview of AADL, STS and PRISM, in Sections 2.1, 2.2 and 2.3, respectively.

2.1 Architecture Analysis and Design Language

The Architecture Analysis and Design Language (AADL) (Feiler et al., 2006) is a textual and graphical language for modeling and analyzing a real-time system's hardware and software architecture as hierarchies of components at various abstraction levels.

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

The AADL core language is designed to be extensible and can be extended via user-defined properties and annex sub-languages. User-defined properties are relatively simpler extensions, when compared to sub-languages, and can be associated with modeling elements as simple values, for instance, integers or strings. However, sub-languages allow more complex structures to be added to an AADL model. A sub-language can be standardized and published as an AADL annex. Several such annexes have been defined, for example, the *behavior annex* to model the component's behaviour, and the *error annex* for modeling the error behaviour of the system. Annex sub-languages are included into AADL specifications as annex libraries or annex subclauses. An annex library is used to define classifiers defined in an anonymous namespace, or in a public or private part of a package. Annex subclauses are inserted into component types and component implementations and can reference the classifiers declared in the annex library. In AADL, annexes are considered to be separate from the core AADL, i.e., if we remove all the annex libraries, subclauses, and annex-related property associations, the resulting model is a valid core AADL model. For further details, the reader can refer to the work (Feiler et al., 2006).

2.2 Stochastic Transition Systems

Stochastic transition systems (STS) (De Alfaro, 1998) are transition systems that support non determinism, and transitions with unspecified delay distributions,

providing concise and compositional means to represent systems in terms of probability, waiting-time distributions, non determinism, and fairness.

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

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

The set of transitions T is partitioned into the set of *immediate transitions* T_i and the set of *delayed transitions* T_d. Immediate transitions must be taken as soon as they are enabled, and a subset of these transitions $T_f \subseteq T_i$ is the set of *fair transitions*. In turn, the set of delayed transitions is partitioned into: (i) the set of transitions with *exponential delay distribution* T_e, where for each $\tau \in T_e$ there is an associated transition rate $\gamma_\tau > 0$, and (ii) the set of transitions with *unspecified delay distributions* T_u that are taken with non-zero delay, but the probability distribution of the delay and the possible dependencies between this distribution and the system's state or past history are not specified.

Given a state $s \in S$, we indicate by $T(s) = \{\tau \in T \mid s \models \varepsilon_\tau\}$ the set of transitions enabled by s. To ensure that $T(s) \neq \emptyset$ for all $s \in S$, an *idle transition* τ_{idle} is added to every STS defined by $\varepsilon_{\tau_{idle}} = true$, $m_{\tau_{idle}} = 1$, $p^{\tau_{idle}}_1 = 1$, $\gamma_{\tau_{idle}} = 1$ and by the set of assignments $\{v' := v\}_{v \in V}$.

2.3 Probabilistic Timed Automata and PRISM

To analyze our multi-agent systems, in this paper we use the PRISM model checker (Kwiatkowska et al., 2002). Among other supported formal notations, PRISM provides symbolic model checking of systems modeled as networks of Probabilistic Timed Automata (PTA), which are semantically described by Timed Probabilistic Systems (TPS)(Norman et al., 2013). De Alfaro shows that an STS can be straightforwardly translated into (fair) TPS, yielding the same state space (De Alfaro, 1998).

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

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

The property specification language of PRISM for PTA is based on Probabilistic Computation Tree Logic (PCTL) (Hansson et al., 1994). The model checker can verify whether the probability of a path property pp is within a bound b, which is specified as: $Pb[pp]$. Here, b can be any of $>= p$, $> p$, $<= p$ or $< p$, where p is a double within [0,1]. A path property pp is a formula that evaluates to either true or false for a single path in the model, in which one can apply the following operators: X (next), U (until), F (eventually), G (always), W (weak until), R (release). PRISM can also compute the minimum and maximum probabilities of a path property, in the form of: $Pmin =?[pp]$, and $Pmax =?[pp]$, respectively. In

order to check a path property for paths that start from multiple states, *filters* are used to identify the starting states. For instance, the "forall" filter returns true if the property is true for all states satisfying the filter.

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

3 A Multi-Agent System Architecture for AAL

Our proposed architecture consists of multiple agents, and ensures improved fault-tolerance, scalability and adaptability, compared to centralized architectures in the domain, such as CAMI (Kunnappilly et al., 2017). The architecture is inspired from similar existing architectures in literature (Tapia et al., 2009). However, existing solutions usually suffer from additional overhead encountered during agent synchronization for collective decision-making and data consistency maintenance. This overhead can sometimes hamper the real-time behavior of the system. Hence, we investigate how we can use these systems for developing integrated solutions that ensure a safe trade off between autonomous behavior and consistency overheads. This is challenging since agents are interdependent, and have only a limited view of the environment. Concretely, the agent-based solution should ensure a consistent view of the environment, in terms of processed data and events, as well as an inter-agent communication overhead that should not result in breaching the real-time system demands. We ensure this by allowing each agent to cater for a particular functionality, respectively; for instance, a health-monitoring agent detects health-parameter variations and raises a notification to caregiver. However, in order for the agents to cooperate in real time, each agent maintains the dependencies it can have with other agents, in a list that can change at run time [1].

The architecture is described briefly in the following, and is shown in Fig. 1. It consists of the following components:

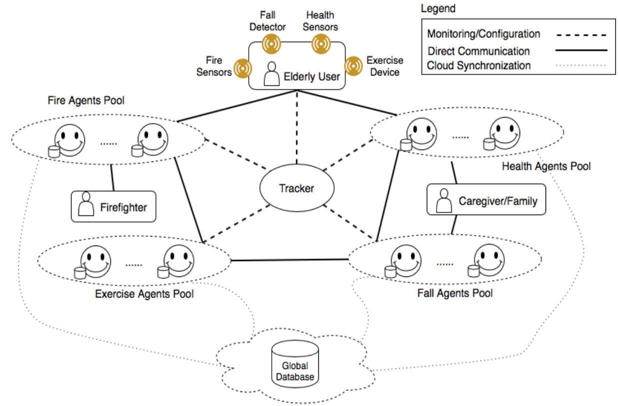

Figure 1: A MAS Architecture for AAL

- **Agents**: In our solution, each agent tackles a particular functionality, in response to the sensor data, that is, the *fire agent* deals with detecting fire events from fire sensors and sends a notification to firefighters, the *pulse agent* detects the pulse data variations and sends a notification to the caregiver, the *fall agent* detects the user fall and alerts the caregiver, the *exercise agent* schedules and monitors the exercise session of the user, etc. These agents can belong to different categories, ranging from simple *reflex* agents to complex *intelligent* agents. In our case, we use the exercise agent as an example of an intelligent agent with embedded reinforcement learning (RL) algorithms. This provides an optimized exercise session for the user, taking into account his/her health condition, preferences and exercise trends. All other agents are modelled as reflex agents encoded using "if-then-else" rules to handle the particular scenario. To be able to cooperate efficiently in real time (by reducing extra overheads), each agent is equipped with a list of possible dependencies with other agents. For instance, a fall agent has a dependency relation with a pulse agent. If a heavy fall occurs, the fall is first communicated to the caregiver, and then the fall agent synchronizes with the pulse agent to see if there are any pulse deviations (like a low pulse). If a low pulse is detected, the fall agent updates its notification to the caregiver indicating that the fall may be due to a low pulse. Each agent also maintains a small local database to store the individual data and keep track of the processed events and the decisions taken. The dependency lists are also maintained in the local database.
- **Tracker**: The system has a tracker that keeps the record of the IP addresses of all the agents in the system. The user's connections to the agents are established via the tracker. If the tracker fails at any point in time, the system continues to function via direct connections between user requests and agents. As shown in Fig 1, we have multiple agents of each category, which can accept requests from multiple users, arbitrarily, based on availability.
- **Cloud Database**: We also maintain a large-scale

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

database in the cloud. All the local databases of the agents eventually synchronize with this cloud database. The cloud database also maintains the domain information about the user, like age, disease history, user preferences, etc.

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

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

3.1 Reinforcement Learning in Exercise Agents

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

$$Utility = w1 * (u_st) + w2 * (ss_com), \quad (1)$$

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

$$u_st = \begin{cases} 1 & \text{if user satisfied} \\ -1 & \text{if user not satisfied} \end{cases} \quad (2)$$

$$ss_com = \begin{cases} 1 & \text{if exercise duration completed} \\ -1 & \text{if exercise duration not completed} \end{cases} \quad (3)$$

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

3.2 Use-Case Scenarios and System Requirements

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

- *Scenario 1: Fall due to a low pulse:* The pulse-detection sensor worn by Jim detects the low pulse, and the wearable fall-sensor detects the fall. The sensors forward the sensed data to the tracker, which assigns a pulse agent and a fall agent to user 1 (arbitrarily, based on availability). The agents communicate with each other and reach the conclusion that the fall is due to a low pulse, and send a notification alert to the caregiver.
- *Scenario 2: Fire and Fall occurring simultaneously:* Mary is cooking dinner, and suddenly she feels dizzy and falls. The cooker is still on, starting a fire at home. In this case, the sensors alert

the tracker of the respective events, and the former assigns a fire agent and a fall agent to Mary. The agents communicate with each other, synchronize the simultaneous occurrences of both events, and alert both the firefighter and the caregiver.

- *Scenario 3: Health abnormality during the exercise session:* Jim gets a calendar notification to start the exercise session. The tracker then assigns an exercise agent to Jim to schedule and monitor the exercise session. The exercise agent communicates with the health agent and identifies that Jim's health is normal and suggests the medium-complexity exercise for cardiac patients based on his preferences and health condition. In the middle of the exercise session, Jim's health agent indicates a sudden increase in pulse and hence the exercise agent suggests an exercise of lower complexity in the next sub-interval. The following system requirements are formulated for the above scenarios:
  - R1: If a fall occurs due to low pulse, then raise an alert to caregiver indicating *fall due to low pulse* within 20 s. It relates to Scenario 1.
  - R2: If a fire and a fall event occur simultaneously, then raise an alert to both caregiver and firefighter indicating the issue, within 20 s. This requirement relates to Scenario 2.
  - R3: The exercise session is scheduled only if the health agent indicates a normal pulse.
  - R4: The initially suggested exercise is based on user preferences and health condition.
  - R5: If any health abnormality is detected in the first sub-session of the exercise, a different set of exercises of lower intensity is prescribed. Requirements R3, R4 and R5 are formulated based on Scenario 3. It should be noted that R1-R5 are safety-critical requirements.

  In addition, the system has quality-of-service (QoS) requirements as follows:
  - R6: If the tracker fails, the system continues its functionality.
  - R7: If one of the agent fails, its function is carried out by the back-up.

4 Modeling Multi-Agent Systems in AADL

In this section, we illustrate the AADL modeling of our MAS, depicted in Fig 1. The components are modeled as follows: the agents and tracker are modeled as abstract components, which can be extended to suit a hardware or software implementation, at later stages of design. The sensors are modeled as hardware devices. The databases are modeled as data components in AADL. All the components have their respective component type and implementation defined. The component type defines the component features and properties. We use bus connections to represent the respective communication protocols used by the components. The bus access is modeled as a feature of the component. We restrict to only using properties like dispatch protocol, period, execution time and user-defined properties to specify the scope of global variables in the system. However, based on the requirements, certain user-defined properties can also be added to specify the rate of occurrence of an aperiodic event or so (Kunnappilly et al., 2018). In the component implementation, we define the sub-components and connections.

Listing 1 shows an excerpt of the AADL model of our system with an exercise agent, and a bus component; the Agent Communication Protocol (ACP) models the communication protocol between multiple agents. For simplicity, we assume that the communication protocols defined here work via shared variables. The *Agent* component is modeled as an abstract component in AADL (Lines 1-8), which can be later refined towards a particular hardware or software, based on the application. We also show a system-level representation (Lines 10-26) with its sub-components, user-defined properties (some of which needs assertion in the respective agent annex, where the property is applied) and their connections defining the communication.

Listing 1: An excerpt of the system modeling in AADL

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

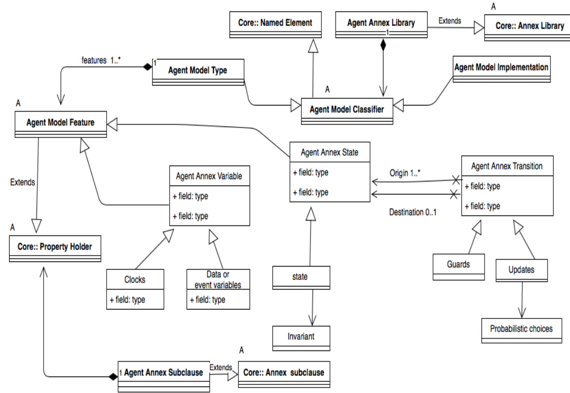

Figure 2: Agent Annex Metamodel

```
26    end agent_system.impl;
```

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

4.1 Modeling Behaviours of Agents in AADL: Agent Annex

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

Metamodel extension of AADL. The structure of our *Agent Annex* is defined by extending the AADL metamodel (Society of Automotive Engineers, 2006), represented as UML2 class diagrams. All classes in the Agent Annex metamodel are defined as subclasses of class *AObject*, the root class of the AADL metamodel. *Named objects* in the Agent Annex model is a subclass of the *Property Holder* class, allowing an object to have a name and associated AADL properties. Abstract classes in the metamodel are tagged by an *"A"*. The *Agent Model Annex* is formulated by extending the AADL abstract classes, *Annex Library* and *Annex subclause*. All the expressions of the Agent Annex are introduced as subclasses of these abstract classes. The *Annex Library* is used to declare *classifiers* of our Agent Annex in packages. The Annex Library concepts are attached to an AADL model by using the *Annex subclause* within a component type or component implementation declaration. An *Annex subclause* can refer to items in the *Annex Library*, and to basic AADL model elements. The Agent Annex metamodel is presented in Fig. 2.

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

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

$$AA = \langle Var, Init, Tt \rangle \tag{4}$$

- where: Var represents the set of local and global variables defined in the AA ;
- $Init$ is the assertion over Var denoting the set of initial states, formulas and also specifies association of reward values to a state or transition ;
- Tt is the set of state transitions, defined accordingly as in Section 2.3.

An excerpt of the Agent Annex subclause of the exercise agent is defined in Listing 2. As shown in Listing 2, the annex defines a probabilistic transition system with 7 states - $Idle$, Op, $Comm$, Exc_sc, $Ex1$, $Ex2$ and $Fail$, and a clock variable x. $Idle$ represents the initial state. It also defines a probabilistic transition from state $Idle$. The transition is enabled aperiodically based on the calendar schedule for exercise and it has a probability of 0.999 to reach the state Op, and of 0.001 to reach the state $Fail$. Lines 11-21 define the other transitions specific to exercise agent. For instance, Lines 11-13 define the transitions for initiating communication with the user's pulse agent. If the agent reaches the $Comm$ state, it will initiate communication with the pulse agent and the exercise session starts only if the user pulse is normal. Lines 16-23 illustrate the exercise recommendation based on RL. Upon reaching the Exc_sc state initially, an exercise recommendation is made to the user based on the user preferences and disease history. The exercise duration is 10 min split in 2 intervals of 5 min each. Upon an initial recommendation (say category 1), the agent moves to the state $Exc1$ (Lines 14-16). The agent stays in this state until the completion of split duration of 5 min or until the user has decided to quit the exercise session. If the exercise split interval is less than 2 and greater than 0 (assuming the duration is 10 min), the exercise schedule has to continue and in the next split interval the agent recommends the action with the highest reward (Lines 20-21). The associated variables and their assertions are defined in the variables section Lines 22-30. Lines 28-30 indicates that the sys_rew1 is associated with the state $Exc1$ depending on user satisfaction or session completion and also with the transition $r1$ defined by lines

17-19. Similarly, there is *sys_rew*2 calculated for exercise 2, however due to space constraints, we do not show transitions for exercise category 2.

Listing 2: An example of Agent Model Annex Subclause attached to Exercise Agent

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

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

5 Formal Encoding of MAS

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

$$AADL_{\text{Comp}} = \langle Comp_{\text{type}}, Comp_{\text{imp}}, AA \rangle, \quad (5)$$

where $Comp_{\text{type}}$ is the component type, $Comp_{\text{imp}}$ represents the component implementation, and *AA*, the agent annex specification.

- $Comp_{\text{type}}$ is in turn defined as a tuple: $Comp_{\text{type}} = \langle Features, Prop \rangle$, where:
  - *Features* model the bus access that abstracts the communication protocol utilized by the system.

AADL comp	STS
$\langle Comp_{\text{type}}, Comp_{\text{imp}}, AA \rangle$	STS
T_{p}	*Invariant + Guard*
T_{e}	*Invariant + Guard*
User-defined properties	*Variables*
Bus	*Variable*
Data	*Variable*
A sub-component	STS
AA states	*Variable values*
AA transitions	*Transitions*
AA variables	*Variables*
AA formulas	*Variable assertions*
AA reward association	*Variable assertions*

Table 1: Encoding of AADL Component as STS

  - *Prop* lists the associated properties of the component, like *Deployment*, *Communication*, *Timing*, *Thread-related properties*, *user-defined properties*, etc. In this work, we only consider a subset of *Timing*, *Thread-related properties*, *user-defined properties* as follows: $Prop = \{T_{\text{p}}, T_{\text{e}}, Dispatch\ protocol\}$, where T_{p} and T_{e} represent the period and execution time of the component, respectively, T_{p}, $T_{\text{e}} \in Timing\ properties$, $Dispatch\ protocol \in \{P, AP\}$, where *P* and *AP* represent periodic activation and aperiodic activation, respectively. *User-defined properties* are used to declare global variables.
- $Comp_{\text{imp}}$ is defined as $Comp_{\text{imp}} = \langle SC, Con \rangle$, where:
  - *SC* represents the sub-components of the system,
  - *Con* represents the set of connections. The function $F_{\text{con}} : Con \rightarrow Features$ assigns *Features* to *Con*.
- Agent Annex *AA* follows the semantics defined in Section 4.1.

Definition 1. *The AADL component defined by Equation (2) is formally encoded as an STS. The MAS architecture is represented as a parallel composition of all the STS modules:* $MAS = ||_{i=0}^{n} STS_module_i$*, where n is the number of AADL components of the system, excluding data components and bus components, if defined in the system. The* STS_module_i *elements are defined as follows:*

- V_l *is defined the set of AA variables that contribute to the state space of* STS_module_i*, local and possibly global variables.. The clock variables values are given by the component's period*

and execution-time properties of $Comp_{type}$ definition. V_g is defined by the global variables that do not contribute to the state of the system.

- *Θ_i denotes assertion over V_l and V_g.*
- *T_i represents the set of transitions defined in the AA of an agent.*

The formal encoding is tabulated in Table 1. We now present an example of the above formal encoding, by applying it on a exercise agent of our use case. The excerpt of the exercise agent (EA) presented in Listings 1 and 2 is formally encoded as an STS module, where:

- V_{EA} (where by EA we denote the exercise agent) : $\{(s1, cal_exc, ss_com, exc_rec, exc_split, x, sys_rew1) \cup (utlity1, w1, w2)\}$
- θ_{EA} : $\{s0 \models (s1 = 0 \wedge cal_exc = 0 \wedge ss_com = 0 \wedge exc_rec = 0 \wedge exc_split = 0 \wedge x = 0 \wedge sys_rew1 = 0), weights \models (w1 = 0.6 \wedge w2 = 0.4), utility \models (utlility1 = w1 * u_st + w2 * ss_com), reward \models ((sys_rew1 = utility1) \cup (sys_rew1 = sys_rew1 + utility1), reward\ association \models (s1 = 4\&(u_sat = 0|u_sat = 1)\&(ss_com = 1|ss_com = 0) : sys_rew1) \wedge ([r1]true : sys_rew1)\}$
- T_{EA} is defined by the transitions as follows [2]:
T_{EA} : $\{\tau1 : \{(s1 = 0 \wedge cal_exc = 1 \wedge ss_com = 0 \wedge exc_rec = 0 \wedge exc_split = 0 \wedge x = 0 \wedge sys_rew1 = 0) \longrightarrow (s1' = 1 \wedge cal_exc' = 1 \wedge ss_com' = 0 \wedge exc_rec' = 0 \wedge exc_split' = 0 \wedge x' = 0 \wedge sys_rew1' = 0), P = 0.999 \cup (s1' = 6 \wedge cal_exc' = 1 \wedge ss_com' = 0 \wedge exc_rec' = 0 \wedge exc_split' = 0 \wedge x' = 0), P = 0.001\},$
$\tau2 : \{(s1 = 1 \wedge cal_exc = 1 \wedge ss_com = 0 \wedge exc_rec = 0 \wedge exc_split = 0 \wedge x = 2 \wedge sys_rew1 = 0) \longrightarrow (s1' = 2 \wedge cal_exc' = 1 \wedge ss_com' = 0 \wedge exc_rec' = 0 \wedge exc_split' = 0 \wedge x' = 0 \wedge sys_rew1' = 0), P = 1\}$

Similarly, all other AADL components are encoded as STS modules, respectively. In the next section, we describe our formal analysis approach with PRISM.

6 System Analysis with PRISM

The STS modules are encoded as a set of PTA modules in PRISM. The architecture is a parallel composition of the PTA modules. Each agent can accept at most 2 connections, and each has a redundant copy. Therefore, in order to ensure parallel processing, we assume 4 PTA for a single category of agent. Thus, we have 16 agent PTA that deal with pulse monitoring, fall monitoring, exercise monitoring and fire monitoring. In addition, we have one tracker PTA, through which connections between the agents are established. The sensor data and internal databases are modeled as variables, and their communication is modeled via shared data access. For simplicity, we have not chosen to model the cloud database.

Listing 3 shows an excerpt of exercise agent encoding in PRISM. Since PTA is a subset of STS, the encoding of STS as PTA modules is a one-to-one mapping, with the syntax adapted to match the PRISM input language. All the global variables and their assertions (weights, utility and rewards) are defined outside the module definition of the exercise agent. Apart from these, the exercise agent module uses a set of local variables. Variable s represent the state, $s = 0$ (Idle), $s = 1$ (Op), $s = 2$ (Comm), $s = 3$ (Exc_sc), $s = 4$ (Exc1), $s = 5$ (Exc2), $s = 6$ (Fail). There are variables that represent the user's calendar exercise input (*cal_exc_u1: [0..1]*), user quit *u1_quit: [0..1]*), session completion *(ss_comp: [0..1]*, where *0* indicates that the event has not occurred, whereas *1* indicates the opposite. There are also variables to represent the exercise split sessions ($exc_split[0..2]$), 0 representing the initial value and 1 and 2 representing the two split sessions respectively, and the exercise recommendations (exc_rec [0..2]) where 0 represent the initial condition and 1 indicating that exercise category 1 is chosen and 2 indicates that category 2 is chosen. Variable x is a clock variable. The invariant associated with the states (Lines 15-17) depend on the component's execution time (defined at the interface of the AADL component's model). The invariant of state *Op* is $x \leq Exec_time$. The transitions defined in Lines 18-27 follow the transitions definition of the Agent Annex specification of the exercise Agent. Finally, in Lines 29-33, we show the association of rewards to the respective states or transitions. After modeling the respective PTA modules, we can perform exhaustive probabilistic verification of the model, and generate probabilistic guarantees for the satisfaction of the functional and QoS requirements listed in Section 3.1.

The verification results are tabulated in Table 2. The requirements are formulated as PCTL queries and the model-checking method is *Digital Clocks*. Since PRISM, by default, returns the value for the (single) initial state of the model while model checking, we employ *filters* to verify our properties over all states. Requirement $R1$ ensures that if a fall event occurs due to a low pulse for $user1$ (Jim), and the tracker is operational, then the tracker initiates the communication between the respective fall and pulse agents associ-

[2] Due to space constraints, we only illustrate the first two transitions according to Listing 2, however all other transitions can be enlisted in a similar way.

ated with user Jim (the request can be assigned to any of the agent sockets depending on availability), and the probability that one of them sends an alert to caregiver indicating that there is "fall due to low pulse" is greater than 0.999 provided that at least one of the sockets of each agent is functional. Assuming that the communication via tracker takes less time, the requirement is satisfied within 10 time units. Similarly, for $R2$, we verify for *user*2 (Mary) that in case of fire and fall events occurring simultaneously, an alert indicating both events is raised and sent within 10 time units, provided that the tracker has not failed. In case of $R3$, $R4$ and $R5$, we verify the functionality of the exercise agent serving Jim. By $R3$, we establish that the exercise session is scheduled only if the corresponding health agent indicates that the user's pulse level is normal. $R4$ indicates that the initial exercise category is chosen based on user preferences and health condition. By verifying $R5$, we show that if a high pulse deviation occurs during the exercise subsession, a low intensity exercise is chosen in the next sub-session, irrespective of user preferences. In $R6$, we illustrate a similar function as in $R2$, but assuming that the tracker has failed. In this case, the functionality is met by direct communication between the agents, which takes more time than the communication via tracker (it is shown that this requirement is satisfied within 20 time units). Next, in $R7$, we assume a fall event of *user*2, and one failed fall agent; then, a fall alert is raised and sent to the caregiver by either one of the redundant fall agents. PRISM shows that this requirement is satisfied within 20 time units.

Listing 3: An excerpt of the PRISM Model of an Exercise Agent

```
1   pta
2   const double w1=1.0;
3   const double w2=1.0;
4   formula utility1 = w1*(u_sat)+w2*(ss_comp);
5   formula sys_rew1=utility1;
6   formula sys_rew1= sys_rew1+utility1;
7   module Exc_agent1
8     s: [0..6] init 0;
9     //states 0 -Idle, 1-Op, 2-Comm, 3-Exc_sc, 4-Ex1,
10    5- Ex2, 6-Fail
11    cal_exc_u1: [0..1] init 0; u1_quit: [0..1] init 0;
12    ss_com: [0..1] init 0; exc_split: [0..2] init 0;
13    exc_rec: [0..2] init 0;
14    x: clock;
15    invariant
16     (s=1 => x<=2)
17    endinvariant
18    [1]s =0 & cal_exc_u1=1 -> 0.999:(s'=1) & (x'=0) +
19    0.001:(s'=6) &(x'=0);
20    [2]s=1  & x=2  -> (s'=2) & (x'=0);
21    [3]s=2  & h_stat_u1=1  -> (s'=3) & (x'=0);
22    [4]s=3  & u1_pref=1 & u1_dis_his=1 &exc_split =0
23    -> (s'=4) & (exc_rec'=1) & (exc_split'=2) & (x'=0);
24  [r1]s=4 &(x=5|u1_quit=1) & h_stat_u1=1 & exc_split
25  < 2 -> (s'=3) & (exc_split'=exc_split+1) & (x'=0);
26  [5] s=3 & exc_split>0 & exc_split<2 & sys_rew1>
27  sys_rew2 -> (s'=3) & (exc_split'=exc_split+1) & (x'=0);
28  endmodule
29  rewards
30  s=4 & (u1_sat=0 | u1_sat=1) &(ss_com=1|ss_com=0):
31  sys_rew1;
32  [r1] true : sys_rew1;
33  endrewards
```

7 Related Work

Modern AAL systems are designed to tackle numerous functions, and to cater for multiple, distributed users, which makes the system design more complex, and calls for design-time formal analysis.

Some related work is directed towards providing formalisms for agents in terms of various logics (Che et al., 2006; Luo et al., 2005). However, some others have proceeded further to develop specification languages/methodologies for agent systems. Some examples include CASL (Shapiro et al., 2002), DESCARTES (Medina and Urban, 2007), etc. These methodologies employ different formalisms, however some of them are complex and are not expressive enough, like in case of CASL. For DESCARTES, tool support for executing the specifications is also provided. Although the approach is promising, the DESCARTES language is still missing constructs to specify adaptive capabilities of agents, nor it provides an analysis framework for MAS. One of the other common approaches, popular in industry also, is the Agent UML (Bauer et al., 2001) one. The approach does not specify the architectural constructs of the system, and lacks formal analysis, unlike the framework that we present in this paper. Few works have considered the specification and formal analysis of agent behavior in architecture description languages (Oquendo, 2004). The AADL-based modeling framework for multi-agent systems, which we propose in this paper, has the benefit of being integrated into a popular framework that also provides tool support.

There are also some approaches that focus on the formal verification of AAL systems. An interesting related work is that of Rodrigues et al. (Rodrigues et al., 2012), who perform dependability analysis of AAL architectures using UML and PRISM. Other interesting research work uses temporal reasoning (Magherini et al., 2013) to formally verify the reliability of AAL systems. However, the above focus only on QoS requirements, and do not look into the critical functions of AAL systems, which require decision making. Unlike these approaches, we carry out our analysis on an agent-based AAL system architecture, focusing on both functional and QoS re-

Req.	Query	Result
R1	$filter(forall, fall_user1 = 1 \& pulse_user1 \leq 50 \& tracker_fail = 0 \rightarrow P \geq 0.999$ $[F((pulse_alert0_u1 = 3 \| pulse_alert1_u1 = 3 \| pulse_alert2_u1 = 3 \| pulse_alert3_u1 = 3)$ $\& (y \leq 10) \& (fall_fail = 0) \& (pulse_fail = 0))]$	satisfied
R2	$filter(forall, fall_user2 = 1 \& fire_user2 = 1 \& tracker_fail = 0 \rightarrow P \geq 0.999$ $[F((firefall_alert0_u2 = 2 \| firefall_alert1_u2 = 2 \| firefall_alert2_u2 = 2 \|$ $firefall_alert3_u2 = 2) \& (y \leq 10) \& (fall_fail = 0) \& (fire_fail = 0))]$	satisfied
R3	$filter(forall, cal_notexc_user1 = 1 \& tracker_fail = 0 \& (pulse_user1 \geq 60$ $\& pulse_user1 \leq 120) \rightarrow P \geq 0.999 \; [F(exc_sch_u1 = 1))]$	satisfied
R4	$filter(forall, exc_sch_u1 = 1 \& u1_disease_history = 1 \& u1_pref = 2$ $\rightarrow P \geq 0.999 \; [F(exc_u1_int1 = 2)]$	satisfied
R5	$filter(forall, exc_sch_u1 = 1 \& interval = 1 \; \& y \leq 5 \; \& pulse_user1 \geq 200$ $\rightarrow P \geq 0.999 \; [F(exc_u1_int2 = 1)]$	satisfied
R6	$filter(forall, fall_user2 = 1 \& fire_user2 = 1 \& tracker_fail = 1 \rightarrow P \geq 0.999$ $[F((fall_alert0_u2 = 2 \| fall_alert1_u2 = 2 \| fall_alert2_u2 = 2 \| fall_alert3_u2 = 2)$ $\& (y \leq 20) \; \& \; (fall_fail = 0) \& (fire_fail = 0))]$	satisfied
R7	$filter(forall, fall_user2 = 1 \& tracker_fail = 0 \& fail1_fall = 1 \; \& fall2_fall = 0$ $\rightarrow P \geq 0.999 \; [F((fall_alert2_u2 = 1 \| fall_alert3_u2 = 1) \& y \leq 20)]$	satisfied

Table 2: Verification results

quirements, and propose a complete modeling and verification framework for distributed AAL systems that involve real-time, fault-tolerant and probabilistic behaviours. As an advantage if compared to another work (Kunnappilly et al., 2018), the verification results obtained with PRISM are exhaustive. In the mentioned work, the authors have proposed a formal assurance framework for AAL system architectures described in AADL, and showed how to verify them in UPPAAL SMC. As different from the work in this paper, the approach assumes a centralized system architecture, and the only probabilistic behaviour considered in the system is component failure, which can occur arbitrarily. In addition, the statistical analysis with UPPAAL SMC, is not exhaustive, but it relies instead on a finite number of simulations.

8 Discussion

This paper presents an architecture for MAS, which we model in the architecture language AADL that we extend with an agent annex intended to model real-time, fault-tolerant and probabilistic behavior of agents, in a unified manner. This approach allows an agent to synchronize only with a limited number of agents in the system (according to its dependency), unlike the traditional case where each agent has to communicate with every other agent in the system to achieve a consistent view of the environment before making a decision (Kunnappilly et al., 2017). We show the design of our MAS architecture applied to AAL domain with 2 agent categories- simple reflex agents, that use *if-then-else* rules and complex intelligent agents that employ learning techniques, like *Reinforcement Learning*. Although we have demonstrated the use-case of Ambient Assisted Living in the paper, the approach fits well for any other applications employing MAS for handling multiple safety critical applications in real-time, e.g., those of automotive systems for which earlier stage analysis is beneficial.

The modeling framework used in this paper is relying on the Architecture Analysis and Design Language (AADL), one of the best-suited architecture description languages to describe real-time embedded systems (Feiler et al., 2006). Although MAS specifications based on logics and domain specific languages do exist and are popular, they are mostly limited to specification of properties at the agent level and also do not have tool support (see Section 7). AADL, on the other hand, allows us to focus on the component level (here *agents*) and also at the system level (*MAS architecture*) and can effectively model agents' real-time characteristics. With our proposed extension to AADL using Agent Annex, a user can also specify the intelligent agent behaviours (which are often probabilistic) and their failures. AADL also supports a graphical plug-in in OSATE tool to visualizethe model and supports analysis with respect to latency, schedulability, resource utilization, etc. (Feiler and Gluch, 2012).

In this work, we encode the semantics of the AADL model and its agent annex as Stochastic Transition Systems. The encoding is suitable due to the probabilistic behaviour of such systems and allows it to be model-checked exhaustively by probabilistic model checkers, like PRISM, or statistically by simulation-based model checkers like UPPAAL SMC. This paper shows a reduced and abstract architecture with only 4 types of agents and hence can be

verified exhaustively with PRISM.

9 Conclusions and Future Work

In this paper, we have proposed an architecture modeling and formal analysis framework for agent-based AAL systems characterized by intelligent, probabilistic, and real-time behaviours. The intelligence is incorporated by using learning algorithms, in our case, the Reinforcement Learning algorithm. The modeling framework is based on one of the well-established architecture description languages for modeling real-time embedded systems, called AADL. As the core AADL does not suffice to represent the probabilistic and non-deterministic behavior of our system, we propose an annex extension to AADL, the so-called Agent Annex that we formally encode as a stochastic transition system. In order to verify a set of critical functional and QoS requirements like timeliness, fault-tolerance etc., we use an exhaustive probabilistic model-checking method, using the state-of-art model checker PRISM.

Our contribution paves the way for the development of formally assured distributed, adaptable, scalable, fault-tolerant systems, with intelligent behaviours and autonomy. The scalability of the proposed framework is supported by the semantic definition of AADL elements that allows an encoding in UPPAAL SMC for instance, for statistical model checking of models that exceed the boundaries of exhaustive model checking. As future work, we intend to extend our architecture with multiple categories of agents and integrate the Agent Annex to the core AADL.

ACKNOWLEDGEMENTS

This work is supported by the EU Celtic Plus /Vinnova project, Health^{5G}- Future eHealth powered by 5G, which is gratefully acknowledged.

REFERENCES

Bauer, B. et al. (2001). Agent UML: A formalism for specifying multiagent software systems. *International journal of software engineering and knowledge engineering*, 11(03):207–230.

Che, H.-y. et al. (2006). A Description Logic Method of Formalizing the Specification of Multi-Agent System. In *Machine Learning and Cybernetics, 2006 International Conference on*, pages 61–65. IEEE.

De Alfaro, L. (1998). Stochastic transition systems. In *International Conference on Concurrency Theory*, pages 423–438. Springer.

Dissaux, P., Bodeveix, J.-P., Filali, M., Gaufillet, P., and Vernadat, F. (2006). Aadl behavioral annex. In *Proceedings of DASIA conference, Berlin*, volume 32.

Feiler, P. H. et al. (2006). The architecture analysis & design language (AADL): An introduction. Technical report, Carnegie-Mellon Univ Software Engineering Inst.

Feiler, P. H. and Gluch, D. P. (2012). *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language*. Addison-Wesley.

Hansson, H. et al. (1994). A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5):512–535.

Kunnappilly, A. et al. (2017). A Novel Integrated Architecture for Ambient Assisted Living Systems. In *41st COMPSAC*, pages 465–472. IEEE.

Kunnappilly, A. et al. (2018). Assuring intelligent ambient assisted living solutions by statistical model checking. In *International Symposium on Leveraging Applications of Formal Methods*, pages 457–476. Springer.

Kwiatkowska, M. et al. (2002). PRISM: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 200–204. Springer.

Luo, J. et al. (2005). Multi-agent cooperation: A description logic view. In *Pacific Rim International Workshop on Multi-Agents*, pages 365–379. Springer.

Magherini, T. et al. (2013). 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.

Medina, M. A. and Urban, J. E. (2007). An approach to deriving reactive agent designs from extensions to the descartes specification language. In *Eighth International Symposium on Autonomous Decentralized Systems (ISADS'07)*, pages 363–367. IEEE.

Norman, G. et al. (2013). Model checking for probabilistic timed automata. *Formal Methods in System Design*, 43(2):164–190.

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

Rodrigues, G. N. et al. (2012). Dependability analysis in the ambient assisted living domain: An exploratory case study. *JSS*, 85(1):112–131.

Shapiro, S. et al. (2002). The cognitive agents specification language and verification environment for multiagent systems. In *Proceedings of 1st international joint conference on Autonomous agents and multiagent systems*, pages 19–26. ACM.

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

Sutton, R. S., Barto, A. G., et al. (1998). Introduction to reinforcement learning. 135.

Tapia, D. I. et al. (2009). An ambient intelligence based multi-agent system for Alzheimer health care. *International Journal of Ambient Computing and Intelligence*, 1(1):15–26.