Politecnico di Milano
Scuola di Ingegneria dell'Informazione

POLO TERRITORIALE DI COMO
Master of Science in Computer Engineering

Design and Implementation of a Siafu-based Simulator for Activities of Daily Living

Supervisor: Prof. Sara Comai
Assistant Supervisor: Hassan Saidinejad

Master Graduation Thesis by: Chiara Galbussera
Student Id. Number: 796143

Academic Year 2014-2015

Abstract

In recent years, the expected lifespan is in constantly growing. This trend has as consequence an important increase of the number of aged people and a large number of requests for health care assistances of elderly people. Current health care system has not enough resources to ensure a satisfactory assistance for a large number of elderly. Sometimes the elderly themselves do not consider the assistance' solution as acceptable because it implies radical changes, like leaving his/her own home, a drastic change in his/her daily routine and a decrease of his/her independence. A different choice can involve the potentialities of Information and Communication Technology to reduce care costs and delay the recovery in a specialized structure, granting a satisfactory well-being level. In order to do this an Ambient Assisted Living (AAL) approach can be adopted transforming the elderly' house in a Smart-home by installing sensors, actuators and interfacing mechanisms. In this way, elderly's life can be monitored in an unobtrusive way and without a radical change in the resident's routine. Nonetheless, the implementation of Smart-homes may have high costs and require a particular attention, during the installation phase, at the type of sensors that should be installed and where they should be installed. The usage of a simulator of Activities Daily Living (ADL) before the installation of the Smart-home can solve these problems. This thesis work presents a Siafu-based simulator (a large scale simulator of ADLs). The simulator aims to reproduce the environment in which the older person lives, the behavior of the dwellers and of their daily activities and the behavior of the various sensors installed in the environment, reaching a simulation close to real situations.

Sommario

Negli ultimi anni le aspettative di vita della popolazione si sono alzate, portando ad un aumento del numero di persone anziane nella popolazione. Questo porta ad avere un numero elevato di richieste di assistenza socio-sanitaria, che non può sempre essere garantita anche a causa dei costi elevati di tale assistenza. Molte volte è la stessa persona anziana a non accettare il "modello" di assistenza che viene proposto in quanto comporta dei cambiamenti radicali, come ad esempio il ricovero in una casa di riposo, che porterebbe l'anziano ad abbandonare la propria casa, le proprie abitudini e la propria indipendenza. Queste soluzioni sono inoltre molto costose e possono portare ad un declino nel benessere della persona anziana. Una soluzione alternativa potrebbe essere quella di sfruttare le potenzialità dell'Information and Communication Technology così da ridurre i costi di assistenza e ritardare l'eventuale ricovero in case di riposo, garantendo il benessere delle persone anziane. Per fare ciò si può usare un approccio Ambient Assisted Living (AAL) rendendo la casa del paziente una Smart-home tramite l'installazione di sensori, attuatori ed interfacce di comunicazione nell'abitazione dell'anziano. Questo approccio non richiede particolari cambiamenti nelle abitudini dell'anziano, la persona non interagisce direttamente con i sensori che vengono installati. Tuttavia l'implementazione di Smart-home ha costi che potrebbere essere elevati e richiede una particolare attenzione, in fase di installazione, al tipo di sensori da utilizzare e alla loro posizione all'interno della casa. Per questi motivi vengono utilizzati dei simulatori di Activity of Daily Living (ADL) prima di procedere con l'installazione. In questo lavoro di tesi viene presentato un simulatore basato su Siafu (un simulatore di ADL su larga scala). Il simulatore mira a riprodurre l'ambiente in cui la persona anziana vive, le tipiche attività che tale persona può svolgere durante la giornata all'interno della propria abitazione e il comportamento dei sensori presenti all'interno della casa per una simulazione che si avvicini il più possibile alla vita reale.

Acknowledgements

I would like to thank my supervisor, Professor Sara Comai and her assistant Hassan Saidinejad for their precious advices, patience and assistance in the development of this work.

I would also like to thank my family and all my friends for their support, their trust in me and their patience during my entire academic path.

Contents

List of Figures **vi**

List of Tables **viii**

1 Introduction **1**
1.1 Problem definition 1
1.2 Siafu simulator overview 3
1.3 Motivation and goals 4
1.4 Thesis organization 5

2 State of the Art **7**
2.1 Agent Based modeling and Simulation (ABMS) 7
2.1.1 Agent definition 8
2.1.2 Review of some existing ABMSs 8
2.2 Smart Environments Simulators 10
2.3 Real life Datasets 12
2.4 Summary of the presented simulators 15

3 Design and Implementation of the Simulator **17**
3.1 Defining the environment of the simulation 17
3.1.1 Background 18
3.1.2 Walls 18
3.1.3 Places 19
3.2 Programming the Behavior 20
3.2.1 Activities and parameters definition 20
3.2.2 The AgentModel 22
3.2.2.1 Agents definition and parameters initialization 22
3.2.2.2 Managing the agent's activities 23
3.2.3 The WorldModel 25
3.2.3.1 Places definition 25

3.2.3.2 Choosing the activity to perform 26
3.2.3.3 Activities definition 26
3.2.3.4 Events planning 27
3.2.4 The ContextModel 28
3.3 Additional feature: usage of sensors in the simulation 29
3.4 Packaging the Simulation 31
3.5 Simulation Output 31

4 Evaluation of the Simulator 33
4.1 Features definition 33
4.2 Features analysis and comparison 34
4.2.1 Analysis in the base version of SIAFU 35
4.2.2 Analysis in the extended version of SIAFU 40
4.2.3 Summarizing the behavior of the two different versions of SIAFU 47
4.3 Limits of the simulator 48

5 Conclusions and Future Work 51
5.1 Conclusions 51
5.2 Future Work 52

Bibliography 53

List of Figures

1.1 Example of living simulation in a city 4
1.2 Example of simulation working life in an office 4

2.1 A typical agent according to Macal and North definition 8
2.2 eHome example of graphical simulation 11

3.1 SIAFU: Background image for the simulation 18
3.2 SIAFU: Walls image (in white) of the simulation 19
3.3 SIAFU: Places image of the simulation 20
3.4 SIAFU: Places image of the simulation 20
3.5 List of possible agent activities 21
3.6 List of constants 21
3.7 Fields of the agent object 22
3.8 AgentModel to create agents and dweller methods 23
3.9 AgentModel handlePerson function 24
3.10 AgentModel handleEvent function 24
3.11 AgentModel eat function 25
3.12 WorldModel create place method 25
3.13 WorldModel doIteration function 26
3.14 WorldModel eat function example 27
3.15 Activities text file 28
3.16 WorldModel start variable definition in planDayEvent 28
3.17 WorldModel end variable definition in planDayEvent 28
3.18 ContextModel overlays example 29
3.19 SensorModel createSensor function 30
3.20 SensorModel handleSensor function 30
3.21 Package of the simulation files 31
3.22 Simulation output example 32

4.1 SCENARIO 1, constants definition 36
4.2 SCENARIO 1, planning an event 36

4.3 SCENARIO 2, staff 6 constants 37
4.4 SCENARIO 2, staff 6 remaining at the meeting 37
4.5 SCENARIO 2, staff 6 goes to the toilet after the end of the meeting 38
4.6 SCENARIO 3, activity execution example 39
4.7 SCENARIO 4, example of binary overlay 40
4.8 SCENARIO 4, example of discrete overlay 40
4.9 Activities text file 41
4.10 Activities interaction 42
4.11 Activities interaction: from parallel to sequence 43
4.12 Hob function 44
4.13 Fridge function 44
4.14 Cook function 45
4.15 PIR sensor example 46
4.16 Switch sensor example 46
4.17 Example of sensors' output 47
4.18 Read function 49

List of Tables

2.1 Smart Home projects 12
2.2 Comparison between Smart Environment Simulators 15
2.3 Comparison between ABMS tools 16

4.1 Features table and scenarios 34
4.2 Comparison with the two version of SIAFU 48

Introduction 1

In our century there is a slow but significant increase of the average age of the world population that will impact several aspects of our lives, [1]. As a consequence, there will be an increment of the difficulty in taking care of elderly people within their own family. For this reason there is an increase in developing care-based technological solution: the smart-environments, like Smart-homes. In order to have a functional smart-home some tools, like Smart Environments Simulators, are needed to design these solutions in a correct way. SIAFU, as we will introduce in Section 1.2, is a simulator that has been developed to generate data for the evaluation and the comparison of different machine learning methods in context-aware settings and it's meant to serve as a way to perform sanity checks, preliminary evaluations and scalability tests before validation with real users [2].

In this chapter we are going to introduce the problem context, the works motivations and goals.

1.1 Problem definition

The implementation of domotic pervasive systems requires some verification and validation before it can be apply. The installation of a prototype in the home of a user can create some issues: study and collection of data requires long periods increasing the development time and the costs, a physical person is needed to install and update the sensors parameters. For these reasons behavior simulators are preferred: they can simulate many data simultaneously.

A behavior simulator can generate in a few minutes a big amount of data, months or years, of one or more person inside the domestic environment.

With this approach it is possible to analyze data quickly and reduce costs and development time.

The main goal of this thesis is to design and develop some improvements of an existing generic large-scale simulator called SIAFU to support a wide range of tasks and scenarios in the complex world of gathering and processing contextual data.

In particular, we would implement a simulator in which we can design a home or a flat environment with more than one room. In the simulation we can have 1 or more persons that live in the house and can move from a room to another one. The persons are involved in different activities during the entire day, in particular we would like to model everyday activities like:

- Make a shower;
- Read a book;
- Relax watching the TV in the bedroom;
- Sink;
- Eat the meal (breakfast, lunch and dinner);
- Sleep during the night;
- Wake up in the morning;
- Seat on a chair in a room;
- Turn on/off the hob;
- Go to the fridge to take something from it;
- Cook a meal.

To simulate an activity several aspects need to be defined: in particular, we need to specify where it would be performed, the time in which it starts and its duration and which agent would be involved in it and then automatically change the agent parameters like the name of the activity that s/he is performing, her/his status and her/his temporary destination.

We would simulate the persons' behavior and the activities in which they are involved for many days in an automatic way. In particular, for each day we would like to memorize different kinds of data:

- Persons' data: for each instant of time of the simulation we would like to know where each person is, in which activity they are involved, their status (that means know if they are seating somewhere or if they are standing in some part of the house);

- Sensors' data: we would like to know the position of each sensors and their status (on or off) for each instant of time of the simulation.

1.2 Siafu simulator overview

SIAFU [2] is a large-scale Ambient Based modeling and Simulation (ABMS) simulator, written in Java and with a pleasant GUI. It is a generic simulation tools for ubiquitous computing. It was developed with the idea to test the functionality of group services and application before performing user evaluations. It make possible to reproduce worlds and scenarios, designing a modular world composed by three elements: the agents, the world and the context. The behavior of the agents, the world and the context are modeled in separate models.

The *agent model*, described in details in Section 3.2.2, is the decision logic of an agent. It decides what an agent should do given it current context and the status of the entities such ad places and other people. It can change the properties of the agent and also set a destination for an agent. The agents are represented as state machines in which the status change is triggered by context switches or by random factors. With SIAFU it is possible to gather context data for each agent, simplifying statistical analysis.

The *world model*, described in details in Section 3.2.3, consists of three parts: the environment that we want to simulate, the places and a global event model that handles events.

The *context model*, described in details in Section 3.2.4, manages the context variables that are used in the simulation. For each variable the context model specifies the possible value of the variable, the model that is used to simulate the values and how the values are distributed over the environment.

There are some implemented examples available on the simulator website [3] like the living simulation of few inhabitants in a city scenario, shown in Figure 1.1 or in smaller contexts such as the working life in an office, Figure 1.2.

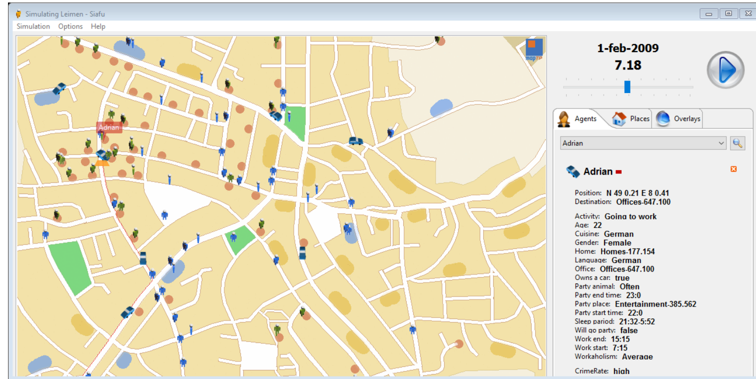


FIGURE 1.1: Example of living simulation in a city

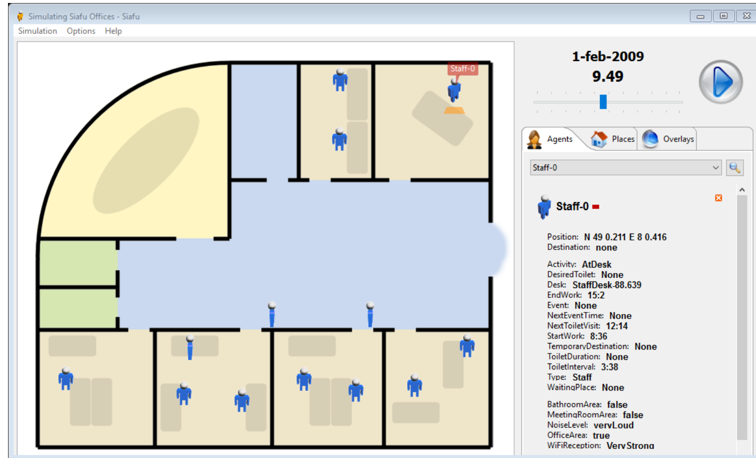


FIGURE 1.2: Example of simulation working life in an office

1.3 Motivation and goals

The simulation of human behavior is a very complex problem caused by its variability. It's difficult to simulate the humans life with a good accuracy, also the routines of people can have unpredictable variations. The attention has been pointed out on a subset of humans events and only on elderly people. We made this choice from the fact that elderly people tend to have more regular life.

The idea to build a simulator like SIAFU stands in the need to test the functionality of group services and applications before performing user evaluations, and we wanted to generate data that are closed to real data.

In particular in this thesis work we are going to:

- Analyze which tools the literature proposes for agent based modeling simulation (ABMS) and for the simulation of smart environments (Smart-home);
- Extend the Siafu simulator in order to handle more complex activities, like nested activities, and in order to have the possibility to introduce and simulate some sensors for a dynamic environment simulation.
- Evaluate the simulator with a features comparison, throw examples, between the base and the extended version of the simulator.

1.4 Thesis organization

The next chapters will focus on the thematics faced during the realization of this work. In particular they are organized as follows:

- Chapter 2 analyzes and discusses the simulation of digital life focusing in particular on two simulators categories: ABMS (Agent Base Modeling Simulators) and simulators of Smart Home Environments. It is also introduced the thematic of the open and free dataset containing sensors domestic recordings.
- Chapter 3 deeply discuss the design and the implementation of the simulator. In particular it focus on the definition of the environment and on the behavior of the various part of the simulation (agents, places and context).
- Chapter 4 discuss and compare our extended version of the simulator and the base one, analyzing some examples of usage.
- Chapter 5 concludes the work with some final remarks and presents some ideas for the future.

State of the Art 2

In this chapter we present the background needed to better understand the topics discussed in this work. In particular, we focus on how the real life can be simulated in a digital way. We are going to analyze the Agent Base Modeling and Simulation (ABMS) tools and the Smart Environments Simulators (that include the usage of sensors in the simulation). In particular, we focus on software solutions that emulate domestic environments, agent-based simulators and general-purpose simulators.

2.1 Agent Based modeling and Simulation (ABMS)

ABMS is a modeling approach that has gained more attention over the past 10 years, as mentioned in Macal et al. work [4], by the increasing number of conferences and presentations and Agent-based software. According to Macal and North [4], ABMS is becoming so popular, thanks to the following factors:

1. Interdependency: ABMS makes possible to model parts of the system as independent agents and simplifies the management of the interdependencies within the whole model;

2. Modularity: with the definition of the behavior of each agent it is possible to simulate more complex systems than with the monolithics approaches. Splitting the complexities over more agents, we can produce more realistic systems;

3. Computational Power: computational power is advancing rapidly, allowing to compute large-scale microsimulation models which were not be possible to model a few years ago.

2.1.1 Agent definition

There isn't a universal definition of the agent. An agent is an independent component within a simulation or a model: an agent is an element that can learn from the environment and dynamically change its behavior with respect to the context. Jennings in et al. [5] says that the most important characteristic of an agent is the autonomy in taking decisions. Macal in [4] joins other characteristics necessary to an agent, shown in Figure 2.1. In particular, an important requirement is the interaction with both the environment and the other agents. The behavior of an agent must depend both on the state in which the agent is and on the context in which he is. These decisions must be taken with feedback systems in order to have a continuous comparison with the results suggests if a choice is good or not.

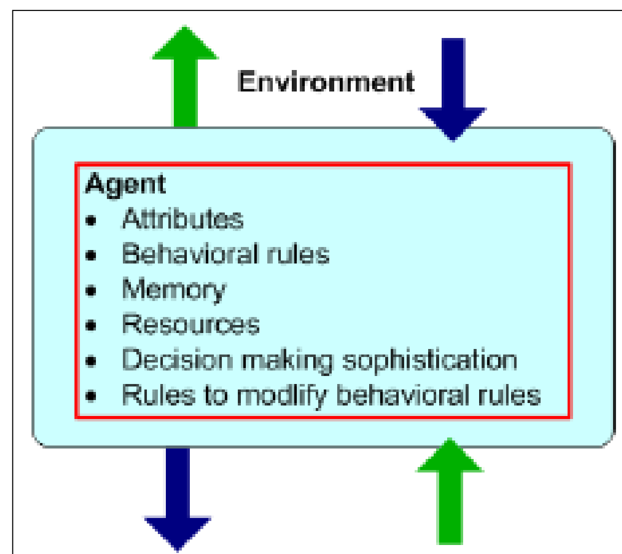


Figure 2.1: A typical agent according to Macal and North definition

2.1.2 Review of some existing ABMSs

According to Macal et al. [6], the ABMS platforms can be divided in two families: General Tools and Specific Tools. A General Tools is able to simulate agents' behavior but also code environmental aspects: are included all the high level programming language like Java, C++ and MATLAB. It

is possible to realize simple and complex systems by implementing some rules. In this group of Tools are included also Spreadsheets with macro programming, for example MS Excel that is probably the simplest approach to model but it produces models with limited agent diversity, restrict agent behaviors and with a poor scalability. In the group of the Specific Tools we have software with specific functions to model the agents, according to the comparison work of Merico et al. [7]. Now we are going to present different meaningful software solutions.

- **MASON**: Multi-Agent Simulator of Neighborhoods [8]. Is a free and open source fast discrete-event multiagent library coded in Java able to handle lightweight simulation tasks and heavy custom-purpose simulations. MASON contains also a model library and visualization tools in 2D and 3D. It was designed to serve ad the basis for a wide range of multi-agent simulation tasks ranging from swarm robotics to machine learning to social complexity environments.
- **SWARM**: Swarm [9] is a kernel and library for multi-agent simulation of complex systems. It was developed at Santa Fe Institute with a focus on artificial life applications and studies of complexity. It's the first ABMS software with hierarchical organization. This software has a conceptual division between the model coding part and the testing part. The advantage of this division consists in the fact that who conduce experiments on the model doesn't need programming skills. SWARM includes also a hierarchical organization in which each context, called swarm, can contains lower levels of swarms which are integrated to the higher level. SWARM is considered the most powerful and flexible simulation platform, but it also have some lacks. It is implemented in Objective-C, because it was designed before the large widespread of Java language, with all its flaws for example requires a good experience with Objective-C and a knowledge of objects orientation paradigm and problems in data area protection. As a consequence, a user can monitor and control any simulation object, no matter how protected it is, also directly from the graphical interface.
- **NetLogo**: is a free and open source software under GPL license available on the web [10]. It is a programmable modeling environment for the simulation of natural and social phenomena, well suited for modeling complex systems. It's written mostly in Scala, a well scalable Object-Oriented language while some parts are written in Java. An example of work developed in NetLogo is the Wolf-Sheep Predation model. In this model two different types of agents are defined, wolves

and sheep. Sheep move randomly around the environment trying to survive, they eat grass and avoid wolves and wolves move randomly around the environment hunting the sheep. NetLogo is a powerful software able to scale with large amounts of agents, but is not configurable and shows flaws when approaching complex models.

- **Repast Simphony**: Recursive Porous Agent Simulation Toolkit [11]. It is a free, cross-platform and open source agent-base modeling toolkit. It was developed at the university of Chicago with a focus on agent bases simulations in social sciences. It is very similar to SWARM providing some libraries for creating, running, displaying and collecting data from simulations. It was born as a Java re-coding of SWARM. Repast, with respect to SWARM, has a better learning curve permitting also to inexperienced users to build complex models. Repast has, according to [12], the greatest functionality's number of any ABMS package: it supports also 2D and 3D view.

- **SIAFU**: is a large-scale ABMS simulator written in Java [2]. It makes possible to reproduce worlds and scenarios, designing a modular world composed by three elements: agents, places and context. The agents are represented as a state machines in which the status change is triggered by context switches. SIAFU makes possible to gather context data for each agent, simplifying statistical analysis.

2.2 Smart Environments Simulators

In this section we are going to analyze some Smart Environment Simulators. The thematic of the simulators is the same but each one has slightly different goals and targets. Some of them have as goal the production of information about users Activity of Daily Living (ADLs), others focus on the low level parts of the system, activating specified sensors inside the environment.

eHome [13], is a project developed in German. A set of advanced domotic services, such as "music-follow-person" and the "conform wake-up" are implemented via hardware. The first one permits to the agent who decides to listen music to move freely among the rooms without turning the speakers on and off. The system monitors the person's position activating the speakers and adjusting the volume according to the room in which the agent is. While the second one, comfort wake-up, computes the optimal wake-up time considering various static and dynamic factors. It analyzes the data provides by the user such as scheduled activities. As a result it tunes home's appliances, like the morning alarm or the coffee machine to be

ready at the right time. The installation inside real house has an important effort therefore they decided to test the system with a software simulator reducing the test time and the cost. eHome is a 2D indoor simulator. It is a point-and-click software. Once started the user can select an agent inside the environment and walk him around rooms interacting with objects like doors, chairs or appliances and the system will properly perform the predefined activities. An example is shown in Figure 2.2.

FIGURE 2.2: eHome example of graphical simulation

ISS (Interactive Smart Home Simulator) [14] is another example of simulator developed in Korea. It is a software able to reproduce a set of rules that defines behaviors on devices installed inside a digital home. The simulator was developed in order to reduce the costs of an hardware implementation of the system. The main goal is a user's tasks automation. The system can reproduce a concrete study case as the following:

> *Currently, this is the autumn season, the most beautiful season in Korea. When Mr. Lee arrives home and after certification, he enters home and the light goes on. Also, the air conditioner is turned on and switched to cooling mode. After changing his clothes, he sits on the sofa. At this time, the context manager, detecting his sitting on the sofa, turns on the TV to a channel based on his favorite. When he goes out to do something, every appliance at home will be turned off.*

The system behavior is based on a set of well-defined if-then-else rules by which every object of the house can change its status. The user cannot be directly interact with the system but a function can extracts randomly an event updating the status of the involved subjects.

TATUS [15] is an environment 3D simulator developed in Dublin. It is able to reproduce smart technological features, like, for example, the identification of people on entry or exit from a room, providing services to authorized persons only. It reproduces via software the context-awareness (the property of pervasive computing to deal with linking change int he environment and computer system) capability of smart devices within rooms.

2.3 Real life Datasets

Simulators and automated domotic systems need a great quantity of data to be tuned. This is often a challenge: high hardware costs, long timing in data gathering, cost of installation, system upgrade and possible faults are issues that should cause some problems. As discussed in [16] two Smart Home dataset categories are identified. The first group is composed by systems with an hight impact and influence in the persons' life: they need the installation of more than one hundred sensors within smart environments. This projects work on the human interaction with Smart environment and objects but they don't focus on healthcare. *Georgia Tech Smart* house [17], *Georgia Tech Aware* home [18] and *PlaceLab* [19] are some examples of software of this category. The second group is composed by low impact systems. A few sensors are used focusing on human activity detection and on health status monitoring. In Table 2.1, according to [16] some data collection projects and their modalities are summarized .

Project	**Multi residents**	**Duration**	**Sensors**	**Activities**	**Occurrences**
ARAS	yes	2 months	20	27	1023-2177
CASAS	yes	2-8 months	20-86	11	37-1513
Kansteren	no	28 days	14	7	245
UvA	no	2 months	14-21	10-16	200-344
Domus	no	11.5 hours	78	0 (user feelings)	NA
Mit	no	2 weeks	77-84	13	176-278

Table 2.1: Smart Home projects

As we can see in Table 2.1, not all the projects are able to handle

more than one agents. We can also identify a broad range of values for the duration of the simulation, this is caused from different aspects:

- A budget limitation: involving different people on the project has high costs and also sensors have their costs;
- Number of sensors: also if the system has a low level impact on the agents' life it can be an elevate number of sensors, the value depends on the size of the smart environment, and on the the number of signals that they want to receive;
- Various different activities: even if all these projects study human life, they consider different activities performed by the agents, varying their number between 7 and 27.

Domus [20]: they asked to the agents to autonomously define their activities to analyze and categorize them offline.

ARAS [21]: Activity Recognition with Ambient Sensing. It is a project for the automatic human ADLs recognition. They are considering two houses with 20 sensors able to capture agents' actions and movements. For each house, ARAS' has recorded a full month of information containing sensor data and activity labels, resulting in a total of two months data. The dataset is available on the website of the project [21].

CASAS [22]: is a project developed in at Washington State University in order to provide an interdisciplinary research platform of intelligent environments. There are two main goals. The first one is the maximization of the user's comfort recognizing, discovering and tracking the user's activities for the automated responses. The second one is to minimize costs, maintenance and saving energy. The datasets are available on the project's website [22].

Kastereen Dataset [23]: the aim of this project was to correctly annotate human' ADLs starting from a great amount of binary data. This experiment left 28 days providing more then 2000 sensor events. Once that they have obtained the data they used two different approaches to annotated the obtained data (Conditional Random Fields and Hidden Markov Model). The dataset is available on the project's website [23].

Contexta-CARE [24]: this project is under development at the Nomadis labs of University of Milan-Bicocca, is an independent Living system able to highlight awareness and ambient intelligence in order to improve the quality of life of users who want to live independently. Awareness is meant as the technological ability to handle and reason on complex sequences of events taking care also the context in which they occur. The system is composed by the Wireless Sensor Network and by the intelligent data-concentrator. The first one is used to collect environmental data (like ambient light, temperature, humidity etc.). The second one processes the data stored in a database, continuously analyzing physical activities carried out by the agents, predicting risky situations. The project has been tested in an independent living environment composed by 4 apartment with 15 agents, proving to be a stable system and collecting necessary data. Their result are not available.

2.4 Summary of the presented simulators

Analyzing the Smart Environments Simulator present in the literature, described in Section 2.2 and summarized in Table 2.2, we can see that the simulators can be too complex to be extended to manage different activities and sensors, like TATUS; or maybe they aren't so difficult to implement but they can't handle complex context information, like ISS, or work with a too restricted set of activities, like eHome. In speech words they are or too complex to implement or they aren't completely compatible with the Agent Based modeling simulation.

	PROS	CONS
eHome	• Handle advance domotic service; • More flexible.	• Need the collection of some data from the user before the simulation start; • Work with a restricted set of activities.
ISS	• Good for user tasks automation; • Based on well-defined if-then-else rule.	• Users cannot interact directly with the system; • Can't handle complex context information.
TATUS	• Reproduce smart technological features; • Deal with context-awareness capabilities of smart devices.	• The implementation is not so easy, it's based on a game engine; • No support real time simulation; • It is like a third person game: the user commands a virtual character to perform tasks.

Table 2.2: Comparison between Smart Environment Simulators

Analyzing the ABMSs present in the literature, described in section 2.1.2 and summarized in Table 2.3, we can understand that some tools have a steep learning curve, like Repast Simphony and SWARM, or have some lack in the documentation and are not so popular to be simply used, like MASON. Netlogo instead is good for complex systems and it is able to scale with a large amount of data but it isn't much configurable, also because written in Scala and not in Java, and it has some flaws with complex model. SIAFU could be a good solution tool for reach our goals, thanks to

its pleasant GUI and the support of multi-agent simulation, but it doesn't support the implementation of complex activities (like nested activities) and sensors. However SIAFU has another interesting features: it designs modular world composed by three elements (agents, places, context). Using this last features we can extend SIAFU in order to have the possibility to model also dynamic environments, throw the usage of sensors, and figured out more complex activities to have a more realistic simulation.

	PROS	CONS
MASON	• Can handle heavy and lightweight simulation tasks; • Support multi-agent simulations.	• Lack in the documentation and popularity.
SWARM	• Support multi-agent simulations; • There is a division from coding and testing part.	• Implemented in Object-C; • Lack in data protection; • Steep learning curve.
Netlogo	• Good for complex systems; • Able to scale with a large amount of data.	• Written in Scala; • Not much configurable; • Flaws with complex models.
Repast Simphony	• It is the Java version of SWARM.	• Not so easy to learn.
SIAFU	• It has a pleasant GUI; • Design modular world composed by three elements (agents, places, context); • Support multi-agent simulation.	• Doesn't support the implementation of complex activities; • Doesn't support the sensors implementation.

Table 2.3: Comparison between ABMS tools

After this analysis we have decided to choose SIAFU for our simulation and we have extended it in order to have the possibility to manage more complex activities and to be able to modeled some sensors.

Design and Implementation of the Simulator 3

This chapter illustrates how the simulator has been designed and implemented, and how a new simulation can be created. We have decided to use SIAFU, a large-scale ABMS simulator able to reproduce worlds and scenarios designing a modular world composed by three components: the agents, the places and the context. SIAFU in able to simulate the behavior of one or more agent (support the multi-agent simulation) that live in house composed by a few rooms. To reach our goals SIAFU isn't "powerful" enough. As a consequence, we need to extend the simulator in order to allow the usage of sensors in the simulation and in order to be able to model more complex activities.

The creation of a new simulation in Siafu is composed by 3 steps:

1. The definition of the environment;

2. The definition of the behavior;

3. The packaging of the various parts of the simulation to make it conforming to a directory structure.

3.1 Defining the environment of the simulation

The simulation environment is composed of 3 parts: the background (e.g., the map of the house), the walls (i.e, the areas that are not accessible), and the places where agents can stay. All of them are created using PNG images.

3.1.1 Background

The background image is simply displayed as the "ground" of the simulation, and is the base of the simulation. The dimension of all the other images created for the simulation must have the same dimension of the background image.

In our simulation the background is the second floor plan of a building of Politecnico di Milano in Como.

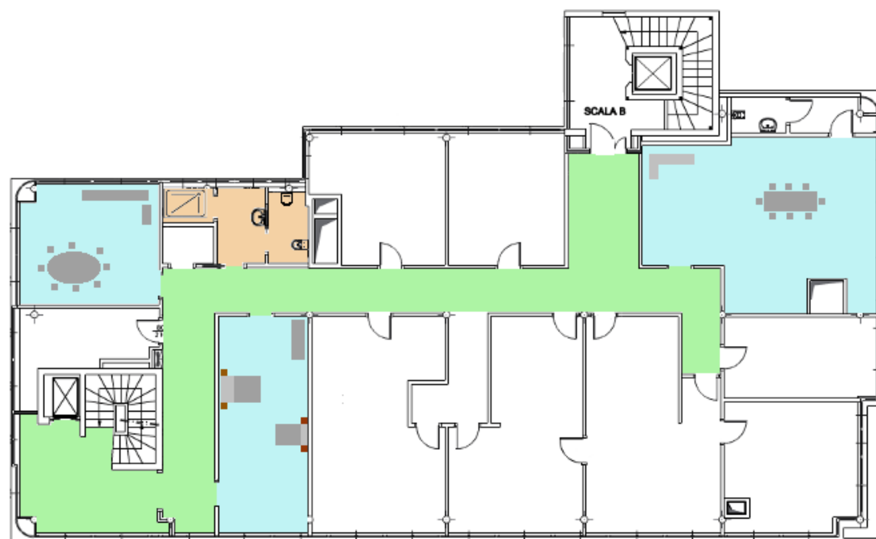


Figure 3.1: SIAFU: Background image for the simulation

3.1.2 Walls

The walls image defines the places where the agents (alias the people in the simulation) can walk and where they can't. This image is based on the background one: walkable areas are marked in black, while the walls and the objects of the room (alias the non-walkable areas) are colored in white.

FIGURE 3.2: SIAFU: Walls image (in white) of the simulation

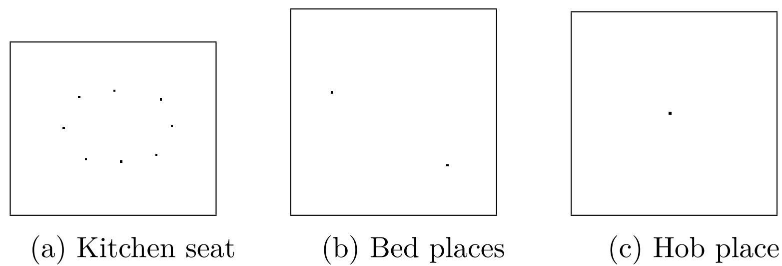
3.1.3 Places

Places define areas where the agents can go or stay. In our case places are used to identify the various activities that an agent can do: for example, if an agent is in the bed (a "*bed place*" will be therefore modeled), it means that the agent is sleeping, or if s/he stay at the "*shower place*" s/he is having a shower.

There are two different ways for the definition of places.

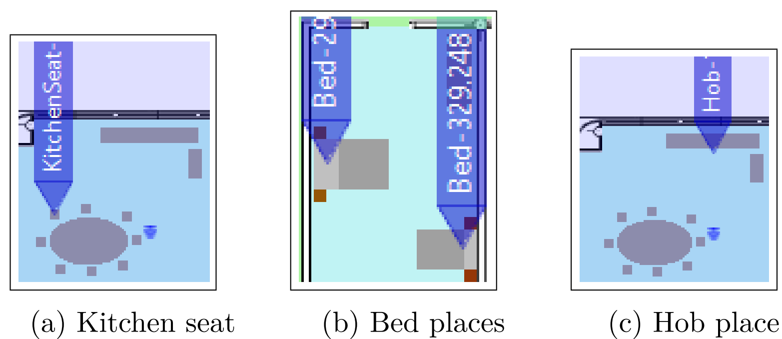
- The first one consists in defining the places in the *WorldModel* (the java class where it is possible to specify what it happens in a place and handle the varoius events that can happen during the day), but it can be very difficult because you need to know the exact coordinates of the places, which are not so immediate.
- A better and simpler way to define the places is by using images: for each type of place one can create an image where every place of that type is represented by a black pixel.

For example, the image in Figure 3.3a represents the places that correspond to the seats around the kitchen's table, 3.3b represents the position of the beds and the Figure 3.3c indicates where the hob is. In the map they are represented like in Figure 3.4.



(a) Kitchen seat (b) Bed places (c) Hob place

Figure 3.3: SIAFU: Places image of the simulation



(a) Kitchen seat (b) Bed places (c) Hob place

Figure 3.4: SIAFU: Places image of the simulation

3.2 Programming the Behavior

To have a running simulation you need to program in java three behavior models: *AgentModel*, *WorldModel* and *ContextModel*. These classes extend the preexisting BaseAgentModel, BaseWorldModel and BaseContextModel of Siafu. There is also a useful class called *Constants*, it isn't mandatory but it helps making the model more readable and "lighter".

3.2.1 Activities and parameters definition

In this section we should describe the definition of the activities in which an agent can be involved. A list of the possible activities is shown in Figure 3.5.

```
46⊖     enum Activity implements Publishable{
47          RESTING("Resting"),
48          GOING_2_SLEEP("Going2Sleep"),
49          IN_TOILET("InToilet"),
50          GOING_2_EAT("Going2Eat"),
51          GOING_2_COOK("Going2Cook"),
52          GOING_2_TOILET("Going2Toilet"),
53          GOING_2_SEAT("Going2Seat"),
54          GOING_2_SHOWER("Going2Shower"),
55          GOING_2_READ("Going2Read"),
56          GOING_2_RELAX("Going2Relax"),
57          GOING_2_SINK("Going2Sink"),
58          AT_SEAT("AtSeat"),
59          ENTERING_TOILET("EnteringToilet"),
60          UNDER_SHOWER("UnderShower"),
61          READ("Read"),
62          RELAX("Relax"),
63          ENTERING_SHOWER("EnteringShower"),
64          AT_SINK("AtSink"),
65          AT_EAT("AtEat"),
66          COOKING("Cooking");
```

FIGURE 3.5: List of possible agent activities

For each activity some parameters should be defined. An example is shown in Figure 3.6. In particular, line 21 indicates when a person wakes up while line 22 indicates that a person goes to the toilet more or less every 3 hours. The Figure 3.7 lists the constants that are displayed in the Agent object: for example, line 26 tells in which activity the person is involved, line 29 indicates if the agent is seating or standing somewhere (corresponding to the value of the variable "*status*"), line 42 tells where the agent is going in that particular moment, line 35 indicates the period of time between two different visits at the toilet, line 33 and 34 tell when the day starts and ends respectively.

```
 9 public class Constants{
10     public static final int INLINE_WANDER = 10;
11     public static final int TOILET_RETRY_BLUR = 90;
12     public static final int MAX_WAIT_TIME = 30;
13     public static final int TOILET_VISIT_DURATION = 20;
14     public static final int DEFAULT_SPEED = 6;
15     public static final int POPULATION = 100;
16     public static final int SENSOR_NUMBER = 10;
17     public static final int TOILET_BLUR = 100;
18     public static final int SLEEP_BLUR = 200;
19     public static final int AVERAGE_AWAKE_HOURS = 16;
20     public static final int WAKE_UP_BLUR = 30;
21     public static final EasyTime AVERAGE_WAKE_UP = new EasyTime(6,30);
22     public static final EasyTime AVERAGE_TOILET_INTERVAL =new EasyTime(3,0);
```

FIGURE 3.6: List of constants

```
24⊖     static class Fields {
25          public static final String STARTPLACE = "StartPlace";
26          public static final String ACTIVITY = "Activity";
27          public static final String TYPE = "Type";
28          public static final String TYPE_S = "TypeS";
29          public static final String STATUS = "Status";
30          public static final String STATUS_S = "StatusS";
31          public static final String SEAT = "Seat";
32          public static final String SENSOR = "Sensor";
33          public static final String WAKE_UP = "WakeUP";
34          public static final String SLEEP = "Sleep";
35          public static final String TOILET_INTERVAL = "ToiletInterval";
36          public static final String NEXT_TOILET_VISIT = "NextToiletVisit";
37          public static final String DESIRED_TOILET = "DesiredToilet";
38          public static final String NEXT_EVENT_TIME = "NextEventTime";
39          public static final String TOILET_DURATION = "ToiletDuration";
40          public static final String WAITING_PLACE = "WaitingPlace";
41          public static final String EVENT = "Event";
42⊖         public static final String TEMPORARY_DESTINATION =
43                  "TemporaryDestination";
44      }
```

Figure 3.7: Fields of the agent object

3.2.2 The AgentModel

The *AgentModel* specifies the behavior of the agents: what they do, where they go, etc.

3.2.2.1 Agents definition and parameters initialization

Now we want to proceed with the creation of the agents that live in the environment that we would simulate.

The first step to do consists in the creation of the agents and their insertion into an ArrayList and in their initialization. For example, in Figure 3.8 the agents are inserted into an ArrayList called *people* (line 66): its size is the value of the constant *POPULATION* defined in the constants class. This method invokes another method, see line 70 in Figure 3.8, that sets all the fields of the agent object. An important thing is that all the agents must have the same fields.

```
65 public ArrayList<Agent> createAgents(){
66     ArrayList<Agent> people = new ArrayList<Agent>(POPULATION);
67     createDweller(people, "Person", "KitchenSeat");
68     return people;
69 }
70 private void createDweller(final ArrayList<Agent> people, final String type,
71     final String seatType){
72     Object[] beds = null;
73     try {
74         beds = world.getPlacesOfType("Bed").toArray();
75     } catch (PlaceTypeUndefinedException e) {
76     throw new RuntimeException("No bed type",e);
77     }
78     Iterator<Place> seatIt;
79     try{ seatIt = world.getPlacesOfType(seatType).iterator();
80     }catch (PlaceTypeUndefinedException e){
81     throw new RuntimeException("No seat type", e);
82     }
83     int i = 0;
84     while(seatIt.hasNext() && i< beds.length){
85         for(int j=0; j < beds.length; j++){
86             Place seat = seatIt.next();
87             bed = (Place)beds[i];
88             Agent a = new Agent(type + "-" + i, bed.getPos() , "HumanBlue", world)
89             a.setVisible(true);
90             EasyTime startDay = new EasyTime(AVERAGE_WAKE_UP).blur(WAKE_UP_BLUR);
91             EasyTime endDay = new EasyTime(startDay).shift(AVERAGE_AWAKE_HOURS, 0)
92                 .blur(SLEEP_BLUR);
93             EasyTime toiletInterval = new EasyTime(AVERAGE_TOILET_INTERVAL)
94                 .blur(TOILET_BLUR);
95
96             a.set(STARTPLACE, bed);
97             a.set(TYPE, new Text(type));
```

FIGURE 3.8: AgentModel to create agents and dweller methods

3.2.2.2 Managing the agent's activities

In order to tell what can do the dweller of the house we implement the functions *doIteration* and *handlePerson* in which you specify the activities that the agents can do and where they can be in a specific period of time of the day. For example, in Figure 3.9, the variable *ACTIVITY* is analyzed. Lines 143-148 show the cooking activity, which specifies the action to do if the agent is in the right place (line 145).With this method it is possible to verify if the agents are at destination, and after reaching the destination it is possible to send them back to where they came from and make them waiting for another activity.

```
129⊖private void handlePerson(final Agent a, final EasyTime now){
130     if(!a.isOnAuto()){
131         return;
132     }
133     try{
134     switch((Activity) a.get(ACTIVITY)){
135         case RESTING:
136             if(now.isAfter((EasyTime) a.get(WAKE_UP)) &&
137                     now.isBefore((EasyTime)a.get(SLEEP))){
138                 goToSeat(a);
139                 a.set(NEXT_TOILET_VISIT, new EasyTime(((EasyTime) a.get(WAKE_UP))
140                     .shift((EasyTime)a.get(TOILET_INTERVAL))));
141             }
142             break;
143         case GOING_2_COOK:
144             if(a.isAtDestination()){
145                 beAtHob(a);
146                 handleEvent(a);
147             }
148             break;
149         case GOING_2_SEAT:
150             if(a.isAtDestination()){
151                 beAtSeat(a);
152             }
153             break;
```

Figure 3.9: AgentModel handlePerson function

Doing an activity for an agent means that he responds to an event that happens at a specific time: this is modeled through the *handleEvent* method that verifies if the agent needs to do something and makes it happen.

For example, in figure 3.10, in line 268 the function checks if the agent needs to eat and makes it happen in line 269.

```
256⊖    private void handleEvent(final Agent a){
257         Object e = a.get(EVENT);
258         if(e.equals(new Text("None"))){
259             return;
260         }
261         String event = e.toString();
262         if(event.equalsIgnoreCase("Shower")){
263             goToShower(a);
264         }
265         if(event.equalsIgnoreCase("Sink")){
266             goToSink(a);
267         }
268         if(event.equalsIgnoreCase("Eat")){
269             eat(a);
270         }
271         if(event.equalsIgnoreCase("Cook")){
272             cook(a);
273         }
274     }
```

Figure 3.10: AgentModel handleEvent function

Each activity corresponds to a different function, defined in the *WorldModel*, and recalled in the *AgentModel*. For example, Figure 3.11 shows the *eat* function setting:

- The place where the activity is done (the destination), line 239;
- The *status* of the agent (if the agent is seated in a place or if he is standing), line 238;
- The activity, line 240;
- The image that represents the agent in the simulation during that action (the color of the image of the agent can change), line 237.

```
236    private void eat(final Agent a){
237        a.setImage("HumanMagenta");
238        a.set(STATUS, a.get(STATUS));
239        a.setDestination((Place) a.get("TemporaryDestination"));
240        a.set(ACTIVITY, Activity.GOING_2_EAT);
241    }
```

FIGURE 3.11: AgentModel eat function

3.2.3 The WorldModel

In the *WorldModel* it is possible to specify what it happens in a specific place and handle the various events that can happen during the day like cooking, eating, having a shower.

3.2.3.1 Places definition

The first thing to do is the creation of all the needed places and the creation of the variable *busy*: this variable identifies if in that place there's an agent or not. Such a method is shown in Figure 3.12.

```
55    @Override
56    public void createPlaces(final ArrayList<Place> places) {
57        for (Place p : places) {
58            p.set("Busy", new BooleanType(false));
59        }
60    }
```

FIGURE 3.12: WorldModel create place method

3.2.3.2 Choosing the activity to perform

Like in the *AgentModel* there is the function *doIteration* to check which event occurred in each period of time. After this point in the *WorldModel* an ArrayList called *times* is created, with all the activities name, start time and end time. In Figure 3.13 at line 76 every element of the *times* ArrayList is analyzed. At lines 77-88 the event *eat* is checked. The variable *now* is compared with the variable that indicates the time in which the meal starts and the variable that indicates if the activity hasn't already been done, line 79. If these conditions are satisfied, the agent can do the activity (line 80) and set the variable *done* at *true* to indicate that the action is done. To terminate the activity the variable *now* is compared with the variable specifying if the time for the meal is ended and then terminates it, lines 85-87.

```
65 public void doIteration(final Collection<Place> places){
66     Calendar time = world.getTime();
67     EasyTime now = new EasyTime(time.get(Calendar.HOUR_OF_DAY), time.get(Calendar.MINUTE));
68     step = now.getTimeInSeconds();
69     if(now.isAfter(NOON) && dayEventsPlanned){
70         dayEventsPlanned = false;
71     }
72     if(now.isBefore(NOON) && !dayEventsPlanned){
73         planDayEvents();
74     }
75     try{
76         for(int i = 0; i<times.size(); i++){
77             if(times.get(i).getName().contains("eat")){
78                 eatEnd = times.get(i).getEnd();
79                 if(times.get(i).getStart() != null && now.isAfter(times.get(i).getStart())
80                         && !times.get(i).isDone()){
81                     start = null;
82                     eat();
83                     times.get(i).setDone(true);
84                 }
85                 if(now.equals(times.get(i).getEnd())){
86                     terminateEat();
87                 }
88             }
89             if(times.get(i).getName().contains("sink")){
90                 if(times.get(i).getStart() != null && now.isAfter(times.get(i).getStart())
91                         && !times.get(i).isDone()){
92                     start = null;
93                     organizeSink();
94                     times.get(i).setDone(true);
95                 }
96                 if(now.equals(times.get(i).getEnd())){
97                     terminateSink();
```

Figure 3.13: WorldModel doIteration function

3.2.3.3 Activities definition

Every activity corresponds to a different function where it is possible to set the variables that are needed to make the activity happen.

For example, to model a person that has to have a meal seated somewhere in the kitchen, you need to check if there are some places of that type and if one of these is free: this is exemplified in Figure 3.14 at lines 236-244. Then the values of the variables of the agent that do the activity can be updated.

For example, in Figure 3.14 at line 249 the value of the variable *EVENT* is set to *Eat*, to indicate that the agent is having a meal; the variable *STATUS* is set to *seated*, line 251, to indicate the fact that he is sitting in one of the kitchen seat; the destination is set to the position of the place where he is seating, line 250.

```
235     private void eat(){
236         Collection<Place> kitchenSeats;
237         try{
238             kitchenSeats = world.getPlacesOfType("KitchenSeat");
239         }catch(PlaceNotFoundException e){
240             throw new RuntimeException(e);
241         }
242         Iterator<Place> kitchenSeatIt = kitchenSeats.iterator();
243         Iterator<Agent> peopleIt = world.getPeople().iterator();
244         int kitchenSeatLeft = kitchenSeats.size();
245         while(peopleIt.hasNext()){
246             Agent p = peopleIt.next();
247             if(kitchenSeatLeft > 0){
248                 kitchenSeatLeft--;
249                 p.set(EVENT, new Text("Eat"));
250                 p.set("TemporaryDestination", kitchenSeatIt.next());
251                 p.set(STATUS, new Text("Seated"));
252             }
253         }
254     }
```

FIGURE 3.14: WorldModel eat function example

3.2.3.4 Events planning

The last step consists in planning every event, i.e. when each event occurs (by specifying some time variables). From a text file called *activities*, structured like in Figure 3.15 we read the name of the activities and all the parameters that we need to calculate the *start* and *end* time variables. Figure 3.16 at lines 384-395 defines the time variable *start* related to the start time of an activity. This variable is defined through some randomly time variables, like in line 385. The parameters for the creation of that random variables are read from the text file mentioned above, lines 380-382. The variable *end*, related to the end of an activity is defined in the same way of the variable *start*, lines 399-413 in Figure 3.17. In lines 414-420 in Figure 3.17 the ArrayList *times* is created and filled.

```
1 cookBreakfast cooking_breakfast_start_hour_min=7 cooking_breakfast_start_hour_max=7 cooking_breakfast_start_minute_max=59 cooking_breakfast_duration_ho
2 eatBreakfast breakfast_start_hour_min=8 breakfast_start_hour_max=9 breakfast_start_minute_max=30 breakfast_duration_hour_max=0 breakfast_duration_minut
3 cookLunch cooking_lunch_start_hour_min=11 cooking_lunch_start_hour_max=12 cooking_lunch_start_minute_max=59 cooking_lunch_duration_hour_max=1 cooking_l
4 eatLunch lunch_start_hour_min=12 lunch_start_hour_max=13 lunch_start_minute_max=59 lunch_duration_hour_max=1 lunch_duration_minute_max=59
5 cookDinner cooking_dinner_start_hour_min=18 cooking_dinner_start_hour_max=19 cooking_dinner_start_minute_max=59 cooking_dinner_duration_hour_max=1 cook
6 eatDinner dinner_start_hour_min=19 dinner_start_hour_max=20 dinner_start_minute_max=59 dinner_duration_hour_max=1 dinner_duration_minute_max=59
7 sink sink_start_hour_min=9 sink_start_hour_max=9 sink_start_minute_max=30 sink_duration_hour_max=0 sink_duration_minute_max=20
8 shower shower_start_hour_min=17 shower_start_hour_max=17 shower_start_minute_max=1 shower_duration_hour_max=0 shower_duration_minute_max=59
```

Figure 3.15: Activities text file

```
380     start_hour_min = Integer.parseInt(line[1].substring(line[1].indexOf(sep)+1));
381     start_hour_max = Integer.parseInt(line[2].substring(line[2].indexOf(sep)+1));
382     start_minute_max = Integer.parseInt(line[3].substring(line[3].indexOf(sep)+1));
383
384     if(start_hour_min == start_hour_max && start_minute_max != 0){
385         start = new EasyTime(start_hour_min, random.nextInt(start_minute_max));
386     }
387     if(start_hour_min != start_hour_max && start_minute_max != 0){
388         start = new EasyTime(random.nextInt((start_hour_max - start_hour_min) + 1)
389                 + start_hour_min, random.nextInt(start_minute_max));
390     }
391     if(start_minute_max == 0){
392         System.out.println("entro if minute == 0");
393         start = new EasyTime(random.nextInt((start_hour_max - start_hour_min) + 1)
394                 + start_hour_min, start_minute_max);
395     }
```

Figure 3.16: WorldModel start variable definition in planDayEvent

```
399 duration_hour_max = Integer.parseInt(line[4].substring(line[4].indexOf(sep)+1));
400 duration_minute_max = Integer.parseInt(line[5].substring(line[5].indexOf(sep)+1));
401
402 if(duration_hour_max != 0 && duration_minute_max != 0){
403     end = new EasyTime(start).shift(random.nextInt(duration_hour_max),random.nextInt(duration_minute_max) + 1);
404 }
405 if(duration_hour_max != 0 && duration_minute_max == 0){
406     end = new EasyTime(start).shift(random.nextInt(duration_hour_max), duration_minute_max);
407 }
408 if(duration_hour_max == 0 && duration_minute_max != 0){
409     end = new EasyTime(start).shift(duration_hour_max, random.nextInt(duration_minute_max) + 1);
410 }
411 if(duration_hour_max == 0 && duration_minute_max == 0){
412     end = new EasyTime(start).shift(duration_hour_max, duration_minute_max);
413 }
414 t = new TimeStamp(name,start,end,false,priority);
415     t.setName(name);
416     t.setStart(start);
417     t.setEnd(end);
418     t.setPriority(priority);
419
420     times.add(t);
```

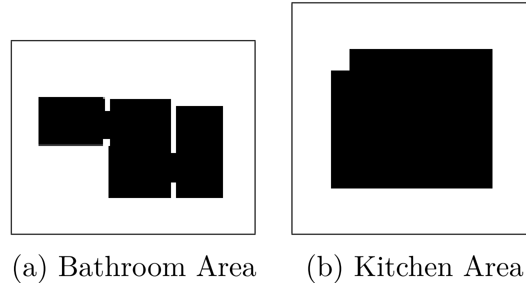
Figure 3.17: WorldModel end variable definition in planDayEvent

3.2.4 The ContextModel

The *overlays*, also called *context variables*, can be used to define contextual variables, like for example the noise level, the temperature at each position in the map, and so on. The *ContextModel* specifies how the overlays change

on over the time. To initialize the overlays we use images, that are translated into a matrix form to allow their modification during the simulation.

In this simulation we use the overlays only to identify the different areas of the map, Figure 3.18: the bathroom area, shown in Figure 3.18a, the kitchen area in Figure 3.18b, the bed room area, and so on.



(a) Bathroom Area (b) Kitchen Area

FIGURE 3.18: ContextModel overlays example

3.3 Additional feature: usage of sensors in the simulation

In our simulation we have also the possibility to introduce and configure some sensors, in particular we are working with PIR sensors and "switch sensors". In order to model the behavior of the sensors a new java class called *SensorModel* is needed.

The *SensorModel* class is similar to the *AgentModel* class described in section 13.2.3.

In particular it contains the functions:

- *createSensor*, shown in Figure 3.19 in which the variables of the object Sensor are set, lines 59-64; then the sensor that we have already created is added to an arrayList that contains all the sensors of the simulation.

- *handleSensor*, shown in Figure 3.20.

  In lines 83-92 the behavior of the PIR sensor is analyzed in particular the sensor is activated when the agent enters in a specific room. To verify this condition we check the variable *context* of the agent and of the sensor; if the result if ***true*** the sensor is activated.

  In lines 95-103 the "switch sensor" is analyzed, in particular the sensor is "on" for the entire duration of the event in which the agent is

involved. In the example in Figure 3.20 the sensor “Hob” is activated as long as the agent is cooking.

```
44    private void createSensor(final ArrayList<Sensor> sensors, final String typeS,
45            final String switchType){
46        Iterator<Place> sensorIt;
47
48        try{
49            sensorIt = world.getPlacesOfType(switchType).iterator();
50        }catch(PlaceTypeUndefinedException e){
51            throw new RuntimeException("No sensor defined", e);
52        }
53        int i = 0;
54        while(sensorIt.hasNext()){
55                Place sensor = sensorIt.next();
56
57                    Sensor s;
58                    s = new Sensor(typeS+"-"+i, sensor.getPos(), "SensorBlue", world);
59                    s.setVisible(true);
60                    s.setPos(sensor.getPos());
61                    s.setDestination(sensor);
62                    s.set(TYPE_S, new Text(typeS));
63                    s.set(STATUS_S, new Text("off"));
64                    s.set(SENSOR, sensor);
65                    sensors.add(s);
66                    i++;
67            }
68    }
```

Figure 3.19: SensorModel createSensor function

```
81      private void handleSensor(final Sensor s, final Object[] agents){
82          try {
83              if(s.getName().contains("Bathroom")){
84                  for(int i = 0;i<agents.length; i++){
85                      Agent a = (Agent)agents[i];
86                      if(a.getContext("BathroomArea").getData().equals
87                              (s.getContext("BathroomArea").getData())){
88                          activateSensor(s);
89                          return;
90                      }
91                  }
92                  disactivateSensor(s);
93
94              }
95              if(s.getName().contains("Hob")){
96                  for(int i=0; i<agents.length; i++){
97                  Agent a = (Agent)agents[i];
98                  if(a.get(EVENT).toString().equals("Cook")){
99                      activateSensor(s);
100                     return;
101                 }
102             }
103             disactivateSensor(s);
104             }
```

Figure 3.20: SensorModel handleSensor function

In addition the function *activateSensor* and *disactivateSensor* are created. These functions are needed to change the *status* and the icon of the sensors. In the first function the *status* is set to "on" and the image is set to this one: , the icon image is green; in the second function the *status* is set to "off" and the image is this one: , the icon image is blue.

3.4 Packaging the Simulation

Siafu is able to run the simulation if it conforms to a directory structure and if it refers to a configuration file. This requires the creation of a folder that contains all the java class needed for the simulation and another folder with all the images in PNG format, shown in Figure 3.21.

The folder that contains the images is organized in some subfolder:

- The subfolder **map** will contain the *background.png* and the *wall.png* images;
- The subfolder **overlays** contains all the overlays images;
- The subfolder **places** encloses all the places images;
- **Sprites** are all the icons that represent the agents(e.g.,) or everything else in the simulation.

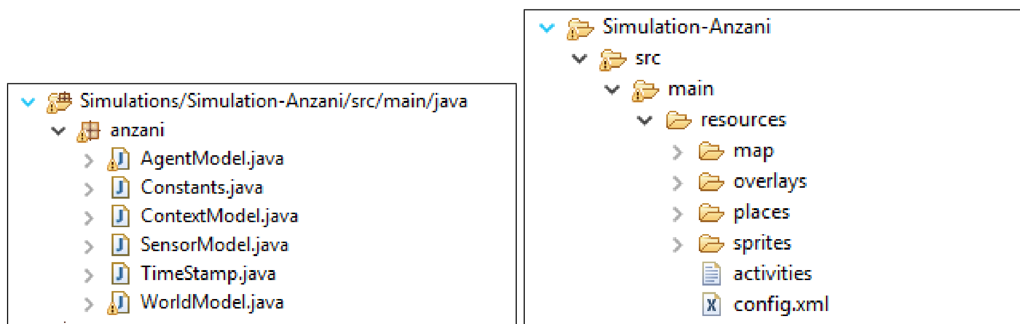


FIGURE 3.21: Package of the simulation files

3.5 Simulation Output

The output of the simulation is an excel file with the values of the variable of interest. In our case the output file, an extract is shown in Figure 3.22, contains these variables and their values:

- ***Time***;
- ***entityID***: the name of agent and sensors;
- ***position***: the coordinates of the agents at that time, and the coordinates of the sensors;
- ***atDestination***: true if the agent is at his destination place, false otherwise;
- ***activity***: the activity that the agent is involved in;
- ***desiredToilet***: indicates the nearest free toilet;
- ***seat***: the place where the agent returns after each activity;
- ***sleep***: the time at which an agent goes to sleep;
- ***event***;
- ***nextEventTime***;
- ***nextToiletVisit***;
- ***WakeUp***: when the agent wakes up in the morning and start his day;
- ***status***: indicates if an agent is standing or seating somewhere;
- ***temporaryDestination***: the place assigned for a certain activity for a certain time
- ***sensor***: indicate if the sensor is "on" or "off";

1	time	entityID	position	atDestination	Activity	DesiredToilet	Event	NextEventTime	NextTo
2	Text:1388552410000	Text:Person-1	de.nec.nle.siafu.model.Position:49.003448177339905#8.006848256157635	BooleanType:true	Text:Resting	Text:None	Text:None	Text:None	EasyTin
3	Text:1388552410000	Text:Person-0	de.nec.nle.siafu.model.Position:49.0034398645 3202#8.006837748768472	BooleanType:true	Text:Resting	Text:None	Text:None	Text:None	EasyTin
4	Text:1388552410000	Text:BedRoomSensor-0	de.nec.nle.siafu.model.Position:49.00344125#8.0068395						
5	Text:1388552410000	Text:BathroomSensor-0	de.nec.nle.siafu.model.Position:49.003442635467984#8.006841251231528						
6	Text:1388552410000	Text:KitchenSensor-0	de.nec.nle.siafu.model.Position:49.0034281650 2463#8.006822960591133						
7	Text:1388552410000	Text:HobSensor-0	de.nec.nle.siafu.model.Position:49.003426933497536#8.006821403940886						
8	Text:1388552720000	Text:Person-1	de.nec.nle.siafu.model.Position:49.003448177339905#8.006848256157635	BooleanType:true	Text:Resting	Text:None	Text:None	Text:None	EasyTin
9	Text:1388552720000	Text:Person-0	de.nec.nle.siafu.model.Position:49.0034398645 3202#8.006837748768472	BooleanType:true	Text:Resting	Text:None	Text:None	Text:None	EasyTin
10	Text:1388552720000	Text:BedRoomSensor-0	de.nec.nle.siafu.model.Position:49.00344125#8.0068395						
11	Text:1388552720000	Text:BathroomSensor-0	de.nec.nle.siafu.model.Position:49.003442635467984#8.006841251231528						
12	Text:1388552720000	Text:KitchenSensor-0	de.nec.nle.siafu.model.Position:49.0034281650 2463#8.006822960591133						
13	Text:1388552720000	Text:HobSensor-0	de.nec.nle.siafu.model.Position:49.003426933497536#8.006821403940886						
14	Text:1388553030000	Text:Person-1	de.nec.nle.siafu.model.Position:49.003448177339905#8.006848256157635	BooleanType:true	Text:Resting	Text:None	Text:None	Text:None	EasyTin
15	Text:1388553030000	Text:Person-0	de.nec.nle.siafu.model.Position:49.0034398645 3202#8.006837748768472	BooleanType:true	Text:Resting	Text:None	Text:None	Text:None	EasyTin
16	Text:1388553030000	Text:BedRoomSensor-0	de.nec.nle.siafu.model.Position:49.00344125#8.0068395						
17	Text:1388553030000	Text:BathroomSensor-0	de.nec.nle.siafu.model.Position:49.003442635467984#8.006841251231528						
18	Text:1388553030000	Text:KitchenSensor-0	de.nec.nle.siafu.model.Position:49.0034281650 2463#8.006822960591133						
19	Text:1388553030000	Text:HobSensor-0	de.nec.nle.siafu.model.Position:49.003426933497536#8.006821403940886						

Figure 3.22: Simulation output example

Evaluation of the Simulator 4

This chapter reports an evaluation study in order to verify the improvements of our version of the simulator. For this purpose, some different possible features of usage are investigated: each feature is analyzed in the base version of SIAFU and in its extended version in order to highlight the differences between the two versions of the simulator. We have decided to *“improve”* the simulator in order to have a more “real” simulation of the typical day of a person that stays for the most of the time in the house, including more difficult activities. In particular we are dealing with sensors that monitor the real position of the person and also to have the possibility to easily understand if the routine of that person changes and if there are some anomalies in his behavior.

4.1 Features definition

To evaluate the functionality of the simulator we have defined four different features to analyze.

- **Feature 1, activities scheduling**: it consists in the definition of the scheduling of the different activities that the agent will do during the day.
- **Feature 2, activity conflict (overlap) management**: it consists in the management of the different activities, i.e. when to do an activity, when to stop it or choose which activity is being to be performed if two of them should start at the same time.
- **Feature 3, activities complexity**: it consists in the “real” execution of the different activities in the simulation.

- **Feature 4, dynamic environment simulation**: it consists in the characterization of the environment of the simulation.

4.2 Features analysis and comparison

In this section we will analyze the different features, described above in section 4.1, in both the version of the simulator in order to shown its capabilities and limits. In order to do this, we can refer to some cases of usage of each feature, Table 4.1.

FEATURE 1	**Scenario 1:** make a shower at 17.00
	Scenario 2: cook breakfast starting between 7 and 8 am
FEATURE 2	**Scenario3:** eat and after that going to the toilet
	Scenario4: randomly choose from relax or read a book
	Scenario5: change activity without ending the first one
FEATURE 3	**Scenario6:** go out from the bedroom and go in the kitchen
	Scenario7: change place during the cooking activity: go to the hob than go to the fridge and return to the hob
FEATURE 4	**Scenario8:** show the overlay for the bathroom area
	Scenario9: usage of sensor

Table 4.1: Features table and scenarios

SCENARIO 1: in this context we are taking into consideration two rooms (the kitchen and the bathroom) and the behavior of only one person. At a predetermined time (5 p.m) the agent moves from the kitchen and goes to the bathroom to make a shower. After the shower he goes back to the kitchen.

SCENARIO 2: in this context we are considering only one room and one agent. The agent that is in the kitchen at a random time (between 7

and 8 a.m) moves from his seat and start cook at the hob. When he has finished, he returns back to his seat and waits for something new to do.

SCENARIO 3: in this scenario we consider the usage of two rooms (bathroom and kitchen) and the behavior of one agent. The agent is eating a meal in the kitchen and he needs to go to the toilet. The agent doesn't interrupt the meal, he first ends it and then goes to the toilet.

SCENARIO 4: in this case we are going to consider two rooms (bedroom and kitchen) and one or two agents. The agents at a certain time (for example 3 p.m) can randomly choose which activity do: read a book lying in the bed or relax in the bedroom.

SCENARIO 5: in this context we are taking into consideration the usage of one room (the kitchen) and the behavior of only one agent. The agent involved in one activity can't change activity as long as the first one is ended (he can't start eating if he hasn't finished to cook).

SCENARIO 6: this scenario consider the usage of the rooms (bedroom and kitchen) and the behavior of one agent. The agent is performing the simple activity of waking up and going to seat in the kitchen.

SCENARIO 7: in this context we are going to take into consideration the usage of one room (the kitchen), the behavior of one agent and the usage of different places in the room. The agent is cooking a meal this means that he need to go to the hob and turns it on, then goes to the fridge and takes some ingredients and returns to the hob to cook them.

SCENARIO 8: in this scenario we are simply going to show the static environment simulation (for example the bathroom area of the map) throw the usage of the overlays.

SCENARIO 9: in this scenario we are introducing the usage of sensors (PIR and "on-off" sensors) and shown their behavior and status during an activity (for example during the cookingBreakfast activity, Figure 4.17).

4.2.1 Analysis in the base version of SIAFU

The base version of SIAFU can model simple simulations with some basic functionalities. The agent can be involved in simple and "predefined" activities. We are going to show this, by analyzing the different features defined

in the previous section.

FEATURE 1: activities scheduling
Referring to SCENARIO 1, the base version of SIAFU permits the scheduling of the different activities during the entire day only in a fixed way, that means that each activity starts and ends according to the decisions of the developer. This means that if we want to plan a meeting during the day, some constant variables, in the *Constants* java file, are must be created. For example in Figure 4.1 we are defining respectively the hour at which the meeting starts, its duration and a possible expected delay of the duration of the meeting. Then in the *WorldModel* java file, shown in Figure 4.2, the event *meeting* is planned using the constants variables previously defined. In this way, every day of the simulation the event *meeting* starts at the same time and has the same duration.

```
34 public class Constants {
35     /** Time for the daily meeting. */
36     public static final EasyTime MEETING_START = new EasyTime(11, 0);
37
38     /** Average duration of meetings. */
39     public static final EasyTime MEETING_DURATION = new EasyTime(1, 30);
40
41     /** Blur of the meeting duration. */
42     public static final int MEETING_DURATION_BLUR = 30;
```

Figure 4.1: SCENARIO 1, constants definition

```
157      /** Plan a meeting at 11h00. */
158      private void planDayEvents() {
159          meetingStart = new EasyTime(MEETING_START);
160          meetingEnd =
161                  new EasyTime(meetingStart).shift(MEETING_DURATION).blur(
162                      MEETING_DURATION_BLUR);
163          dayEventsPlanned = true;
164      }
```

Figure 4.2: SCENARIO 1, planning an event

FEATURE 2: activity conflict (overlap) management
Referring to SCENARIO 3, the base version of SIAFU has only one method to manage the occurrence of two activities in which the second one starts when another one is being acting: ignore the second activity as long as the first one is ended. For example, an agent is at the meeting event that starts at 11.00 while has a duration of one hour and a half plus a possible delay,

shown in Figure 4.1, and an agent needs to go to the toilet at 11.53, like *“staff 6”* in Figure 4.3. In Figure 4.4 we can see that for the agent *“staff 6”* is arrived the time to go the toilet but he stays at the meeting and go to the toilet only when the meeting is ended, shown in Figure 4.5.

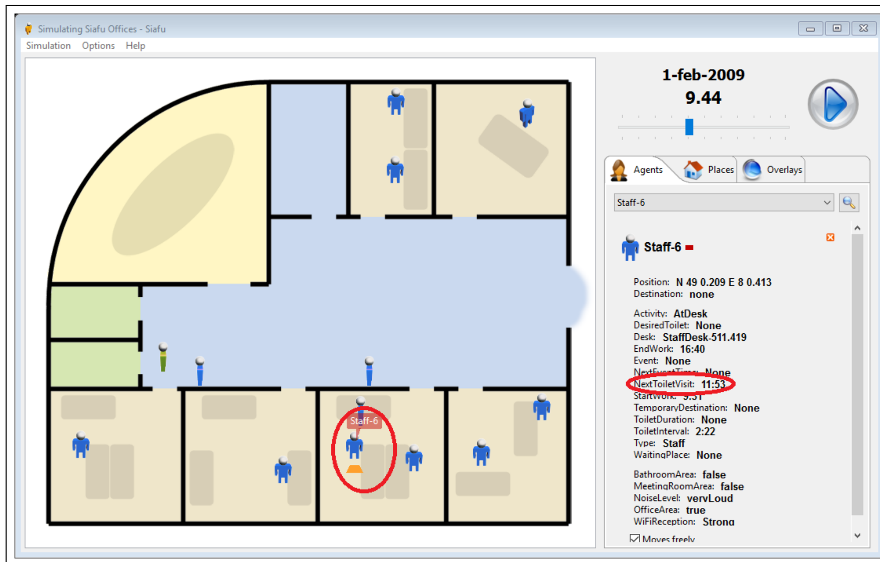


FIGURE 4.3: SCENARIO 2, staff 6 constants

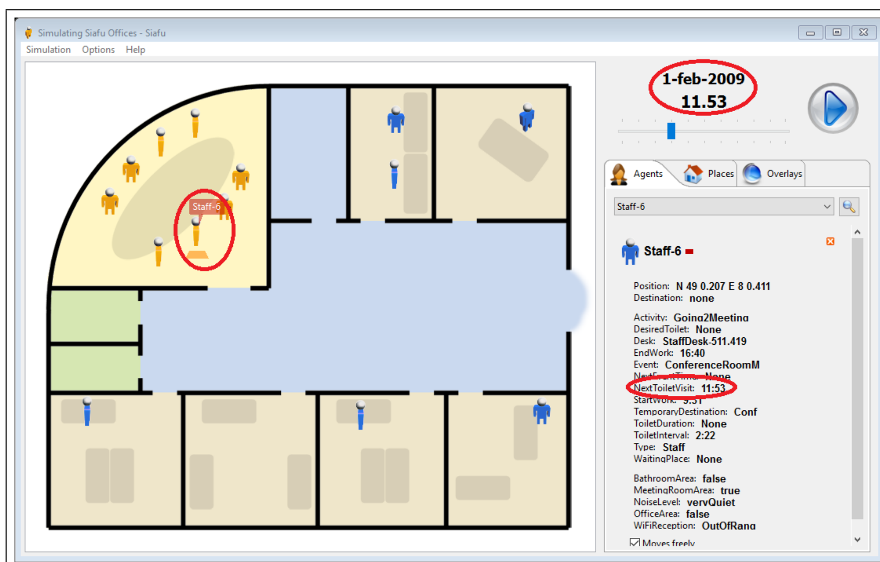


FIGURE 4.4: SCENARIO 2, staff 6 remaining at the meeting

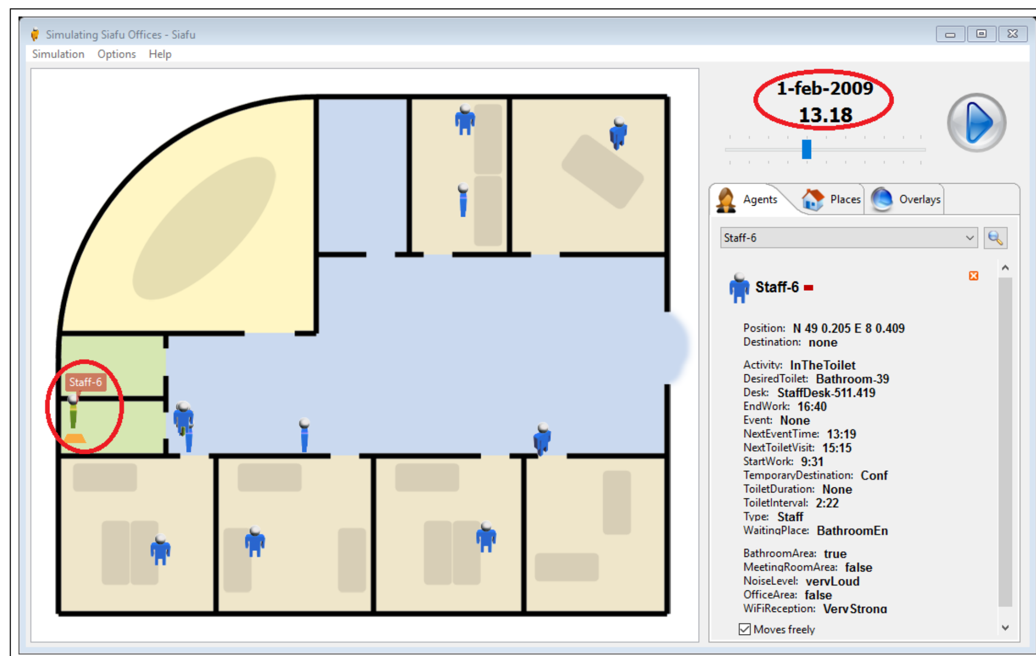
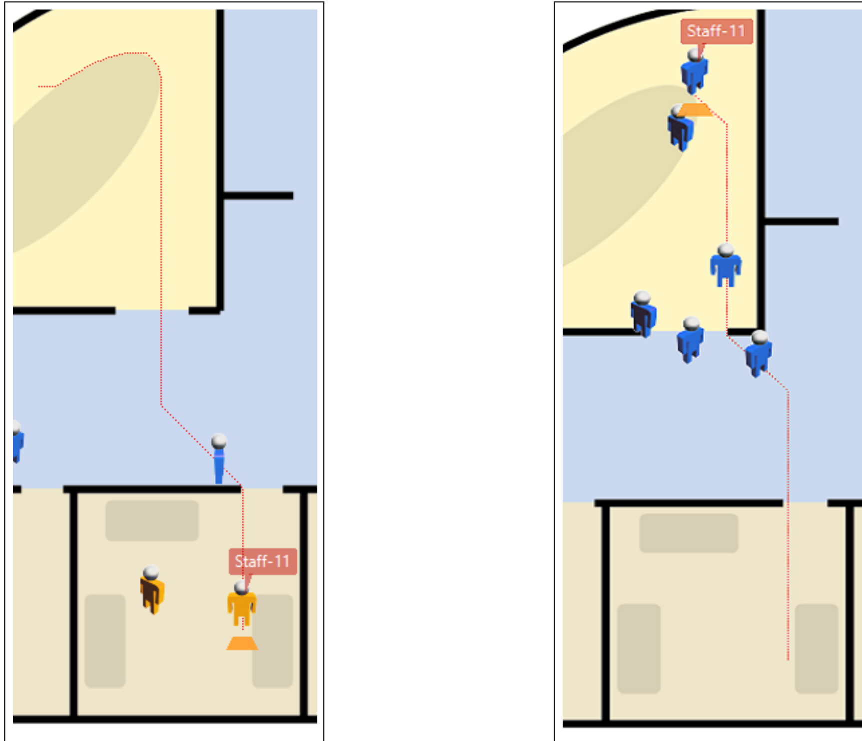


FIGURE 4.5: SCENARIO 2, staff 6 goes to the toilet after the end of the meeting

FEATURE 3: activities complexity
Referring to SCENARIO 6, the base version of SIAFU is able to handle only simple activities. We define simple activity the one that consist in change the place were the agent is only when the activity starts and returns to the place where he stays for his default activity. In particular we have two default activities, one for the night and one for the day when nothing else happens: ***RESTING***, during the night (the agents are sleeping in the bed), ***AT_SEAT***, during the day (when the agents haven't nothing of particular to do, they are sitting on a seat in the kitchen).

An activity starts when the time of the current simulation is equal to the start time of the activity; in that moment the agent moves to the place associated with the activities. The agent remains in the place of the activity as long as the activity ends and than he goes back to his place. In the Figure 4.6 the path that the agent follows in order to go from his desk to the seat in the meeting room and the opposite path to return to his desk are shown.



(a) Agent going to the meeting

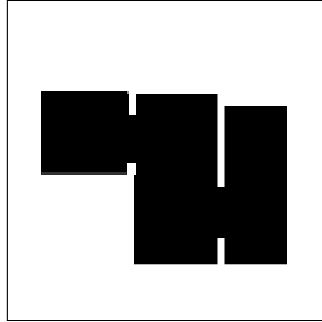
(b) Agent returning to his desk

FIGURE 4.6: SCENARIO 3, activity execution example

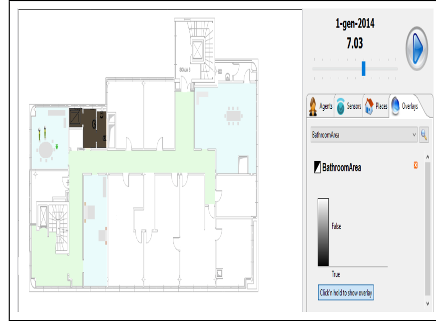
All the activities in this version of SIAFU work like we explain before. Activity that change place more then once can't be handle in this version of the simulator. It also isn't able to deal with activity that should starts at the same time for the same agent.

FEATURE 4: dynamic environment simulation
Referring to SCENARIO 8, the base version of SIAFU describes the environment throw a static behavior: the usage of overlays. Two different types of overlays are used:
binary overlays, used to identify the different parts of the map, and **discrete overlays**, used to identify for example the noise level or the WIFI hotspot range in each position in the map. That's In the first type any pixel value above a threshold translate as true, otherwise it translates to false. In the second type are define multiple thresholds and tag for each interval of value; the pixel value is then fit on the right one. For each overlay an image in gray scale is created. In Figure 4.7 there is an example of binary overlays: the image used to create the *"BathroomArea"* overlay,

Figure 4.7a, the result of that in the simulation, figure 4.7b.



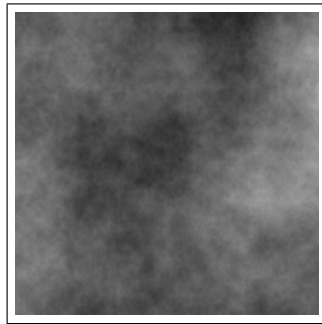
(a) BathroomArea overlay image



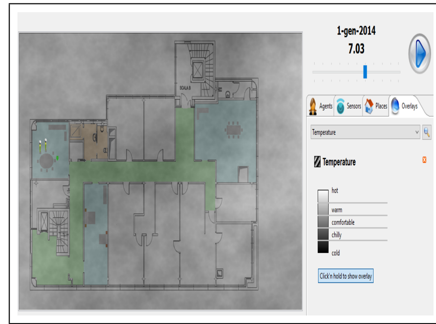
(b) OfficeArea overlay result

Figure 4.7: SCENARIO 4, example of binary overlay

In Figure 4.8 there is an example of discrete overlays: the image used to create the *"Temperature perception"* overlay, Figure 4.8a, the result of that in the simulation, Figure 4.8b.



(a) Temperature perception overlay image



(b) Temperature perception overlay result

Figure 4.8: SCENARIO 4, example of discrete overlay

4.2.2 Analysis in the extended version of SIAFU

In this version of SIAFU we have improved the basic function of SIAFU, like the possibility of create random activities, manage the activities that occur in the same time in a better way and the possibility for the agent to be involved in complex activities like nested activities (activities that are formed by different simple activities). We have also introduced the usage of some sensors in order to have a better and complete simulation and to

discover if there are some "anomalies" or significant changes during the routine of the day.

FEATURE 1: activities scheduling Referring to SCENARIO 2, in this version of SIAFU the activities can be schedule in a fixed way, like in the base version of the simulator, or in a random way: that means that the variables that are needed, like the start time of the activity and the duration of it, are randomly selected between a range of possible values with a fix minimum and maximum. This values are written in a txt file called *"activities"*. The file has a fixed structure, shown in Figure 4.9. Each line corresponds to a different activity and to be correctly read by the simulation the variables must be written in the following order:

1. The name of the activity;
2. The minimum hour at which the activity can start;
3. The maximum hour at which the activity can start;
4. The maximum minute at which the activity can start;
5. The maximum hour for the duration of the activity;
6. The maximum minute for the duration of the activity.

Each variable must be separate from the next one by one white space and when the variable is a number it must be declared as $name = number$, (for example, $breakfast_start_hour_min = 9$).

```
1 cookBreakfast cooking_breakfast_start_hour_min=7 cooking_breakfast_start_hour_max=7 cooking_breakfast_start_minute_max=59 cooking_breakfast_duration_ho
2 eatBreakfast breakfast_start_hour_min=8 breakfast_start_hour_max=9 breakfast_start_minute_max=30 breakfast_duration_hour_max=0 breakfast_duration_minut
3 cookLunch cooking_lunch_start_hour_min=11 cooking_lunch_start_hour_max=12 cooking_lunch_start_minute_max=59 cooking_lunch_duration_hour_max=1 cooking_l
4 eatLunch lunch_start_hour_min=12 lunch_start_hour_max=13 lunch_start_minute_max=59 lunch_duration_hour_max=1 lunch_duration_minute_max=59
5 cookDinner cooking_dinner_start_hour_min=18 cooking_dinner_start_hour_max=19 cooking_dinner_start_minute_max=59 cooking_dinner_duration_hour_max=1 cook
6 eatDinner dinner_start_hour_min=19 dinner_start_hour_max=20 dinner_start_minute_max=59 dinner_duration_hour_max=1 dinner_duration_minute_max=59
7 sink sink_start_hour_min=9 sink_start_hour_max=9 sink_start_minute_max=30 sink_duration_hour_max=0 sink_duration_minute_max=20
8 shower shower_start_hour_min=17 shower_start_hour_max=17 shower_start_minute_max=1 shower_duration_hour_max=0 shower_duration_minute_max=59
```

FIGURE 4.9: Activities text file

FEATURE 2: activity conflict (overlap) management
Referring to SCENARIO 4, in the "extended" version of the simulator it might happen that during an activity another one needs to be perform by the same agent. In particular we can have 3 different case of interaction between different activities, shown in Figure 4.10 in which the black line indicates the activity **A** and the blue line indicates the activity **B**.

(a) Case 1 (b) Case 2 (c) Case 3

Figure 4.10: Activities interaction

In **case 1** where activity **B** starts after the end of the activity **A** (Figure 4.10a) there aren't any problem in the management of the two activities because the agent completes the first activity and then performs the second one, that's the normal behavior for the activity in the base version of the simulator.

In **case 2** where activity **B** starts before the start of the activity **A** and ends after its end (Figure 4.10b) the simulator can deal with the two activities different ways:

1. The agent can ignore the second activity and continues the first one;

2. The agent can stop doing the first activity and start from the beginning the second activity;

3. The agent can finish the first activity and then performs the second one but not from the beginning time. For example **A** is scheduled to start at 9.00 and end at 9.30 and *B* to start at 9.25 and end at 9.40 for a duration of 15 minutes. In this case the agent finishes activity **A** and then performs **B** but with a duration of 10 minutes and not of 15 minutes.

Also in **case 3** where activity **B** starts after the start of **A** and ends before the end of **A**, shown in Figure 4.10c, the simulator can assume different behaviors:

1. Completely ignore activity **B** and do the activity **A** like in the simple case;

2. Start **A** and stop it when **B** is ready to be performed, then perform **B**;

3. Start the first activity, stops it in order to perform the second activity. When the second activity ends return to the first one to complete it for the remaining time.

In both **case 1** and **case 2** there is a particular behavior: when the agent is performing an activity and he needs to go to the toilet the simulator ends the activity in which the agent was involved and lets him go to toilet. This means that the activities are not in parallel but are in sequence, like in Figure 4.11.

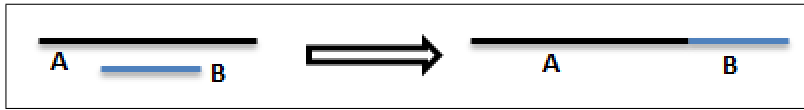


FIGURE 4.11: Activities interaction: from parallel to sequence

FEATURE 3: activities complexity
Referring to SCENARIO 7, in the implemented version of the simulator we can easily manage simple activities, like in the base version as described in FEATURE 3 in Section 4.2.1, and also activities that are more complex. A complex activity that we have analyzed is relative to the cooking a meal. This activity is tagged as *"complex"* because it consists in some consequent steps with different movements in different places without changing the activity (in the base version a change of place stands for a change of activity). In the cooking activity, for example, the first thing that the agent does consists in going to the hob and turn it on; then he may need to take something from the fridge and then return back to the hob in order to put the ingredients in the pan and maybe repeat these actions more than once. This behavior is implemented with nested activities. The first thing to do is writing the different functions that correspond to the sub-activities that are part of the complex activity, in our example the simple activities are the *"hob"* function, shown in Figure 4.12, and the *"fridge"* function, shown in Figure 4.13. Then, the activity *"cook"* consists in calling such functions, as in Figure 4.14.

```
private void hob(){
    Collection<Place> hobs;
    try{
        hobs = world.getPlacesOfType("Hob");
    }catch(PlaceNotFoundException e){
        throw new RuntimeException(e);
    }
    Iterator<Place> hobIt = hobs.iterator();
    Iterator<Agent> peopleIt = world.getPeople().iterator();
    int hobLeft = hobs.size();
    while(peopleIt.hasNext()){
        Agent p = peopleIt.next();
        if(hobLeft > 0){
            hobLeft--;
            p.set(EVENT, new Text("Cook"));
            p.set("TemporaryDestination", hobIt.next());
            p.set(STATUS, new Text("Up"));
        }
    }

}
```

FIGURE 4.12: Hob function

```
private void fridge(){
    Collection<Place> fridges;
    try{
        fridges = world.getPlacesOfType("Fridge");
    }catch(PlaceNotFoundException e){
        throw new RuntimeException(e);
    }
    Iterator<Place> fridgeIt = fridges.iterator();
    Iterator<Agent> peopleIt = world.getPeople().iterator();
    int fridgeLeft = fridges.size();
    while(peopleIt.hasNext()){
        Agent p = peopleIt.next();
        if(fridgeLeft > 0){
            fridgeLeft--;
            p.set(EVENT, new Text("Cook"));
            p.set("TemporaryDestination", fridgeIt.next());
            p.set(STATUS, new Text("Up"));
        }
    }
}
```

FIGURE 4.13: Fridge function

```
if(times.get(i).getName().contains("cook")){
    int stepStart = times.get(i).start.getTimeInSeconds();
    cookEnd = times.get(i).getEnd();

    if(times.get(i).start != null && now.isAfter(times.get(i).start)
            && !times.get(i).isDone()){
        hob();
    }

    if(step >= stepStart+120 && step < stepStart+300){
        fridge();
    }

    if(step >= stepStart+420 && step < stepStart+600){
        fridge();
    }

    if(now.equals(times.get(i).end)){
        terminateCook();
        times.get(i).setDone(true);
    }
}
```

FIGURE 4.14: Cook function

FEATURE 4: dynamic environment simulation
Referring to SCENARIO 9, the introduction of some sensors allow us to describe the environment in a dynamic way because we can know where the agent is and what he is doing at any time thanks to the status of the sensors. In particular two different types of sensors are introduced:

1. The sensors for the presence of the agent in a room: context driven.

2. The “switch” sensors: activity driven.

The sensors of the first type by default are set to *off* and they remain in that state as long as an agent enters in the room where the sensor is placed, in that case the status change to *on*. When the agent goes away from the room the sensor returns to the *off* state. To understand when the agent and the sensor are in the same room we perform a context analysis: the sensor turns *on* if and only if its context and the context of the agent have the same value (the context indicates the area of the map where the agent or the sensor are in a certain moment). This is checked in the function *handleSensor* in the *SensorModel* java class, the Figure 4.15 shown the check of the context for the bathroom presence sensor.

```
private void handleSensor(final Sensor s, final Object[] agents){
    try {
        if(s.getName().contains("Bathroom")){
            for(int i = 0;i<agents.length; i++){
                Agent a = (Agent)agents[i];
                if(a.getContext("BathroomArea").getData().equals
                        (s.getContext("BathroomArea").getData())){
                    activateSensor(s);
                    return;
                }
            }
            disactivateSensor(s);

        }
```

Figure 4.15: PIR sensor example

Also the sensors of the second type have as the state set to *off* as default, but in this case they remain in that state as long as the agent starts the activity in which the sensor is involved and turns it *off* only when the activity is ended. For example, the activity *cook* implies that the agent goes to the hob and cooks something: the sensor related to the hob turns *on* when the agent starts cooking and turns *off* when he ends the activity. Also this check is performed in the function *handleSensor* in the *SensorModel* java class. An example is shown in Figure 4.16.

```
if(s.getName().contains("Hob")){
    for(int i=0; i<agents.length; i++){
    Agent a = (Agent)agents[i];
    if(a.get(EVENT).toString().equals("Cook")){
        activateSensor(s);
        return;
    }
}
disactivateSensor(s);
}
```

Figure 4.16: Switch sensor example

The big difference between the two types of sensors stands in the fact that the second one remains activated also when the agent aren't physically in the same place of the sensor. That is the reason that permits us to implement and perform complex activities, like nested activities, without

having an anomalous behavior of the sensor during the execution of the activities.

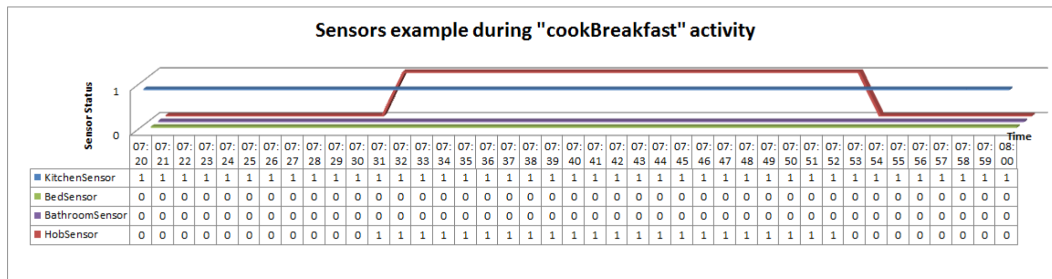
4.2.3 Summarizing the behavior of the two different versions of SIAFU

As we can see from Table 4.2 the two versions of the simulator can be involved in simple activity. In particular both can:

- manage activities that start and end in a preset time (scenario 1);
- wait to do an activity unless the first one is ended (scenario 3);
- change the place where the agent is staying (scenario 6);
- show during the simulation the overlay of a specific area of the map (scenario 8);

The extended version of the simulator can also:

- chose the start time and the duration of an activity in a random way in a specific interval of values defined by the user (scenario 2);
- select which activity to perform among those having the same start time (scenario 4);
- deal with complex activities that include the change of different places during its execution (scenario 7);
- use sensors in the simulation and analyze their behavior (scenario 9). In Figure 4.17 there is an example of the output of the sensors during the activity *cookBreakfast*;



	07:20	07:21	07:22	07:23	07:24	07:25	07:26	07:27	07:28	07:29	07:30	07:31	07:32	07:33	07:34	07:35	07:36	07:37	07:38	07:39	07:40
KitchenSensor	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
BedSensor	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
BathroomSensor	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
HobSensor	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1

	07:41	07:42	07:43	07:44	07:45	07:46	07:47	07:48	07:49	07:50	07:51	07:52	07:53	07:54	07:55	07:56	07:57	07:58	07:59	08:00
KitchenSensor	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
BedSensor	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
BathroomSensor	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
HobSensor	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0

FIGURE 4.17: Example of sensors' output

		SIAFU	
		Base	Extended
FEATURE 1	**Scenario 1:** make a shower at 17.00	YES	YES
	Scenario 2: cook breakfast starting between 7 and 8 am	NO	YES
FEATURE 2	**Scenario3:** eat and after that going to the toilet	YES	YES
	Scenario4: randomly choose from relax or read a book	NO	YES
	Scenario5: change activity without ending the first one	NO	NO
FEATURE 3	**Scenario6:** go out from the bedroom and go in the kitchen	YES	YES
	Scenario7: change place during the cooking activity: go to the hob than go to the fridge and return to the hob	NO	YES
FEATURE 4	**Scenario8:** show the overlay for the bathroom area	YES	YES
	Scenario9: usage of sensor	NO	YES

Table 4.2: Comparison with the two version of SIAFU

4.3 Limits of the simulator

Neither the base version nor the extended version of SIAFU can resolve the case in which there are two activities that have a partial time overlap and the second activity starts when the first one is still in execution. In that case the simulator ignores the second activity and continues the first one.

A big limit of the simulator consists in the fact that we can specify only one type of place in which an activity can be perform, we can't associate two different place's types for one activity. For example the activity *read* can be perform in places of type *bed*, line 276 in Figure 4.18, that means that the agent can't perform the activity *read* in place that aren't *bed*, he can't read seat on a chair because chair and bed haven't the same *place type*.

```
261    private void read(){
262        Collection<Place> beds;
263        try{
264            beds = world.getPlacesOfType("Bed");
265        }catch(PlaceNotFoundException e){
266            throw new RuntimeException(e);
267        }
268        Iterator<Place> bedsIt = beds.iterator();
269        Iterator<Agent> peopleIt = world.getPeople().iterator();
270        int bedLeft = beds.size();
271        while(peopleIt.hasNext()){
272            Agent p = peopleIt.next();
273            if(bedLeft > 0){
274                bedLeft--;
275                p.set(EVENT, new Text("Read"));
276                p.set("TemporaryDestination", bedsIt.next());
277                p.set(STATUS, new Text("Seated"));
278            }
279        }
280    }
```

FIGURE 4.18: Read function

Conclusions and Future Work 5

5.1 Conclusions

In this work we have presented a simulator that is able to handle a multi-agent simulation in a smart home environment. It serves as a way to perform preliminary evaluations and scalability tests before validation with real users and real smart home. The ambient that has been simulated is specified by four information sources: an agent model, a context model, a world model and a sensor model. The latter has been introduced in this thesis. With this clear separation between the different information sources, the simulator can easily simulate a wide range of different scenarios.

We have focused on modeling these information sources and on how they are handled by the simulator. In particular, the ambient that we have modeled is a house in which two elderly persons live and in the simulator we handle the activities they do during the day.

We have improved different features of the simulator in order to obtain a more flexible and complete simulator able to handle more complex activities and a dynamic environment.

In particular:

- The activities can be schedule in a fixed way or in a random way (the start time and the duration of the activity is randomly select between a range of values);

- The simulator can manage the conflict or overlap between different activities in a better way deciding when to do the activity, when to

stop it and choose which activity is being to be performed if two of them should start at the same time;

- Now we can handle also simple activities, a simple movement from a place to another one, and complex activities that consist in some consequent steps with different movements in different places without changing the activity;

- Our simulator, with the introduction of some sensors, is able to describe the environment in a dynamic way because we can know where the agent is and what he is doing at any time thanks to the status of the sensors.

5.2 Future Work

The simulation of human behavior is a complex task, from this work it is possible to identify some interesting further developments in order to improve the simulation's power.

One of these improvements could be the possibility, for the simulator, to recognize particular activities like the absence of the person from the house, for example when the dweller is out during dinner time (currently we model activities that are always at home), the simulator might recognize that he is eating outside. Moreover the set of activities simulated can be improved with other activities (ADLs) involving all the rooms of the house and not only one or two rooms. For example the possibility to clean the entire house or listen the music also if the person is not in the same room of the radio.

Another interesting improvement should be the inclusion of more sensors in order to have more parameters, like the weather condition and the sunrise/sunset time, to increase the simulator realism.

One more important topics is the possibility to perform simultaneous actions to ensure more realistic life simulation, for example the possibility to watch the TV during a meal or listen the music when a person is reading a book.

Bibliography

[1] CDC. Public health and aging: Trends in aging — united states and worldwide. In *MMWR 2003; 52(06);101-106*, 2003.

[2] Miquel Martin and Petteri Nurmi. A generic large scale simulator for ubiquitous computing. In *Mobile and Ubiquitous Systems: Networking and Services, 2006 Third Annual International Conference on*, pages 1–3. IEEE, 2006.

[3] Siafu simulator website. `http://siafusimulator.org/`, 2015.

[4] Charles M Macal and Michael J North. Agent-based modeling and simulation. In *Winter simulation conference*, pages 86–98, 2009.

[5] Nicholas R Jennings. On agent-based software engineering. In *Artificial intelligence, 117(2):277-296*, 2000.

[6] Charles M Macal and Michael J North. Tutorial on agent-based modeling and simulation. In *Proceedings of the 37th conference on Winter simulation*, pages 2–15, 2005.

[7] Davide Merico and Roberto Bisiani. An agent-based data-generation tool for situation-aware systems. In *Intelligent Environments (IE), 2011 7th International Conference on*, pages 129–134. IEEE, 2001.

[8] Mason project website. `http://cs.gmu.edu/~eclab/projects/mason/`, 2015.

[9] Swarm project website. `http://www.swarm.org`, 2015.

[10] Netlogo website. `https://ccl.northwestern.edu/netlogo/`, 2015.

[11] Repast simphony project website. `http://repast.sourceforge.net/`, 2015.

[12] Grids abms comparison. `http://www.grids.ac.uk/Complex/ABMS/`, 2015.

[13] Ibrahim Armac and Daniel Retkowitz. Simulation of smart environments. In *Pervasive Services, IEEE International Conference on*, pages 257–266. IEEE, IEEE, 2007.

[14] Jin Gook Kim Tam Van Nguyen and Deokjai Choi. Iss: the interactive smart home simulator. In ICACT 2009, editor, *Advanced Communication Technology, 2009. 11th International Conference on*, volume 3, pages 1828–1833. IEEE, 2009.

[15] Eleanor O'Neill, Martin Klepal, David Lewis, Tony O'Donnell, Declan O'Sullivan, and Dirk Pesch. A testbed for evaluating human interaction with ubiquitous computing environments. In *Proceedings of the First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, TRIDENTCOM '05, pages 60–69. IEEE Computer Society, 2005.

[16] Ozlem Durmaz Incel Hande Alerndar, Halil Ertan and Cem Ersoy. Aras human activity datasets in multiple homes with multiple residents. In *Pervasive Computing Technologies for Healthcare (PervasiveHealth), 2013 7th International Conference on*, pages 232–235. IEEE, IEEE, 2013.

[17] Hicham El-Zabadani Jeffrey King Youssef Kaddoura Sumi Helal, William Mann and Erwin Jansen. The gator tech smart house: A programmable pervasive space. In *Computer*, pages 38(3):50–60, 2005.

[18] Brian Jones ED Price Elizabeth D Mynatt Julie A Kientz, Shwetak N Patel and Gregory D Abowd. The georgia tech aware home. In *CHI 08 extended abstracts on Human factors in computing systems*, pages 3675–3680. ACM, 2008.

[19] JS Beaudin Jason Nawyn E Munguia Tapia Stephen S Intille, Kent Larson and Pallavi Kaushik. A living laboratory for the design and evaluation of ubiquitous computing technologies. In *CHI 05 extended abstracts on Human factors in computing systems*, pages 1941–1944. ACM, 2005.

[20] Domus website. `http://domuslab.fr/`, 2015.

[21] Aras project website. `http://netlab.boun.edu.tr/WiSe/aras/`, 2015.

[22] Casas dataset website. `http://ailab.wsu.edu/casas/datasets/`, 2015.

[23] Kasteren project website. `https://sites.google.com/site/tim0306/datasets`, 2015.

[24] Roberto Bisiani Davide Merico and Hashim Ali. Demonstrating contexta-care: A situation-aware system for supporting independent living. In *Pervasive Computing Technologies for Healthcare (PervasiveHealth), 2013 7th International Conference on*, pages 309–310. IEEE, 2013.