A Simulator for Generating and Visualizing Sensor Data for Ambient Intelligence Environments

Mario Buchmayr *, Werner Kurschl b, Josef Küng c

*Research Center Hagenberg, 4232 Hagenberg, Austria
bUpper Austria University of Applied Sciences, 4232 Hagenberg, Austria
cInstitute for Application Oriented Knowledge Processing, 4020 Linz, Austria

Abstract

The development of ambient intelligence environments has become a complex task and faces several challenges due to improved sensing possibilities and new fields of application, like detecting activities of daily living or classifying situations. Without having representative test data for experimentation during the development phase, the choice of suitable sensors and classification mechanisms becomes a critical issue. From an economical point of view, hardware and test labs are expensive and require lead time before test data are generated. To compensate the temporal gap, data needs to be simulated until real data are available for testing. In this paper we present an adaptive and extensible simulator which can be used during the design phase to generate data from various sensors within a simulated ambient intelligence environment. Our approach allows generating and visualizing different sensor data as well as simulating faulty or unexpected sensor behaviour.

Keywords: Simulation Environment, Simulator, Activity Recognition, Ambient Intelligence

1. Introduction and Motivation

Ambient Intelligence originated from the field of smart and pervasive environments [1], [2] and evolved to a paradigm for intelligent environments which support or assist a resident based on discovered environmental knowledge. Nowadays, ambient intelligence environments are supposed to be more than simple home automation systems. Especially, when they are used for situation or activity detection they
can utilize different off-the-shelf sensors and devices to perceive contextual information. Based on the requirements for an ambient intelligent home, a system architect has to cope with several issues: i) choosing the right sensors which provide sufficient environmental and contextual information, ii) choosing a proper classification and/or reasoning approach, iii) gathering test or training data, iv) evaluating the results, and v) verify if additional sensor input or different reasoning approaches would improve or impair the result.

Usually all these decisions and activities must be settled before the sensor devices are installed. From a software development point of view, the lack of test data is a critical issue in many cases. Test data are needed at an early project stage to verify if the intelligence component can detect the required use cases with the available set of sensor data. In case of supervised learning approaches pre-classified test data are needed to train the system before it is possible to perform any classification. If unsupervised learning approaches are applied, test data are needed to verify if the use case descriptions are sufficient for a proper detection.

Another issue is to get test data which are similar to data perceived from a real environment. This means sensor data which contains disturbing noise or wrong sensing results from a faulty sensor. Sometimes it is necessary to simulate irrational sensor input caused by a user behaviour which does not correspond with the behaviour assumed during the use case definition. Especially in projects where new sensing hardware is developed in parallel to the software or test labs are not available, obtaining sensor data is a critical issue.

There are two principle approaches to solve the test data problem. One solution is to create a laboratory which contains the used sensors or sensor prototypes for producing data. Laboratories are a good but costly and time consuming solution, especially when sensor hardware needs to be developed first. A parallel development of sensor hardware and system software is only possible with restrictions. The second solution is the usage of simulation environments. Simulation environments allow arranging various sensors on different building blueprints and produce data by triggering sensor interaction on behalf of the user. Many sensor manufacturers provide sample simulation environments for their platforms, mostly to demonstrate the applicability of their product. Unfortunately, these tools are not generic and only support a restricted set of sensor types and fixed smart home layouts used for demonstration purposes. A further disadvantage is that these tools are not intentionally designed for simulating user interactions within a whole ambient intelligence environment, including the possibility to simulate different users and user behaviours.

Therefore, we propose the usage of a generic simulation environment during the design and development phase of an ambient intelligence environment. The environment must support an extensible set of sensors and allow i) the generation of simulated sensor data for activity or situation detection, and ii) the visualization of generated or sensed data. The benefit of this methodology is that insufficient system designs can be detected at an early development stage and cost-benefit-estimations for expensive sensor hardware can be done in advance. The simulation approach is suggested in addition to systematic evaluations under real life conditions which are still indispensable.

2. Related Work

During the research for this paper we investigated different tools for designing and simulating smart homes and sensor equipped environments. Most of them are not primarily intended for simulating and visualizing heterogeneous sensor data in ambient environments. They were rather used for testing new intelligent devices or usability studies [3]. To ease the evaluation of possible sensor environments we decided to classify them in two categories: i) 3D-based and ii) 2D-based environments.

3DSim [3] is a tool for three dimensional-based rapid prototyping of ambient intelligence applications. It can be used for prototyping smart devices for meeting places, like lights, display walls, projectors and
aware chairs. One of the main functionality is the dialogue management which enables the usage of gestures for pointing to devices and objects. 3DSim focuses on evaluating usability issues of new smart devices. Another 3D-based simulation tool is USEd. The User Scenario Editor (USEd) [4] from the Czech Technical University follows a more data driven approach for testing applications in ambient intelligent environments. The main purpose is to transform video recorded data into a virtual environment for further evaluation. This concept is quite feasible for testing and evaluating the usability of new devices, but cannot be used for our intended simulation purpose.

A 2D-based tool which focuses ‘on controlling and simulating the behaviour of an intelligent house’[5] is the ISS (Interactive Smart Home Simulator). It visualizes the interaction between a resident and its environment and provides a set of home automation sensors. A further tool for two dimensional modelling is the CASS (Context-Aware Simulation System) which generates ‘context information associated with virtual sensors and virtual devices in a smart home’[6] with the intention to use the gathered data for demonstrating self adaptive applications. Both tools, the ISS and the CASS, abstract from concrete hardware and allow a set of heterogeneous sensors. But neither the ISS nor the CASS provide a plug-in concept which allows adding user defined sensors. Furthermore, both tools lack the possibility to intercept the sensed data or to simulate corrupted sensors.

3. Requirements

Based on an investigation of different tools for modelling ambient systems we have defined a set of essential requirements for a simulator intended to be used for generating and visualizing sensor data. One major requirement (R01) for the simulator is to provide an exchangeable floor plan layout of a flat which can be equipped with different sensors. Although three dimensional based simulators allow a more detailed distinction of actions, because of the possibility to detect vertical movements (for example useful for detecting falls), we decided for the simpler 2D approach for two reasons: i) most of the floor plans are available in 2D only, and ii) it facilitates the development of new sensor items which are placed on the floor plan. This leads to the second requirement we defined. The simulator must support a component or plug-in architecture which allows to hook-in new user defined sensor simulator items into the existing environment (R02). It must be possible to place these sensor items freely on the floor plan and produce simulated data by performing user interaction on these items (e. g. touching or clicking them with the mouse). For demonstration purpose a set of common home automation sensors, such as contact sensors, switch sensors or motion sensors must be provided.

An important function we identified and which is currently not supported by any other simulation environment we evaluated, is the possibility to produce manipulated, false or faked sensor data. This functionality is useful if it is necessary to simulate disturbing signal noise or corrupt sensor behaviour. A faulty sensor behaviour which occurs commonly in ambient environments is that a sensor sends signals without being activated. Therefore, we require the possibility to apply different data manipulation steps on a sensor item (R03) and to randomly trigger sensor signals without user interaction (R04). A further requirement for the simulator is that all generated simulation data must be provided in real-time via an output-interface (R05) to support forwarding the data to an intelligence component for further processing.

The last requirement (R06) for the simulator is that it must have an interface for visualizing incoming sensor data on the user interface. Providing this functionality the simulator can be used to visualize sensor input from a real system or to replay pre-recorded sensor data. By using the replay functionality in combination with the real-time output-interface it is possible to use the simulator for comparing different reasoning or classifying approaches based on the same pre-recorded data. Therefore, an objective comparison of different classification or reasoning approaches can be guaranteed and the system’s behaviour can be visualized which fosters a better understanding. Table 1 gives a brief summary of the mentioned requirements.
Table 1. Summary of the requirements for the simulation environment

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R01: Adaptable Floor Plans</td>
<td>Support for different two dimensional floor plan layouts</td>
</tr>
<tr>
<td>R02: Pluggable Sensors</td>
<td>Pluggable sensor items for simulating sensor data</td>
</tr>
<tr>
<td>R03: Data Manipulation</td>
<td>Filters for manipulating sensor data (adding signal noise)</td>
</tr>
<tr>
<td>R04: Corrupted Sensors</td>
<td>Possibility to simulate corrupt sensors (random or unexpected sensor signal)</td>
</tr>
<tr>
<td>R05: Real-Time Output</td>
<td>Provide interfaces for getting generated data at real time</td>
</tr>
<tr>
<td>R06: Visualization Interface</td>
<td>Interfaces for visualizing incoming sensor data (real or simulated sensor data)</td>
</tr>
</tbody>
</table>

4. Architecture

The architecture of our simulator follows a component-driven approach and is mainly based on the architectural MVC pattern [8]. It consists of the following components:

- **Environment.** The environment contains basic configuration settings for the simulator. During start up available sensor views are registered by the environment and corresponding controllers are created for the registered views. Afterwards signal filters, signal handlers and signal parsers for processing the sensor signals are added.

- **Sensor Signal.** The sensor signal encapsulates the data sensed by a sensor. It represents the internal state of a sensor and can be compared with the *model* in the MVC pattern. Each sensor has to define its own sensor signal. Sensor signals are created by the sensor view and distributed in the whole simulator by the sensor controller.

- **Sensor View.** The sensor view is the graphical user interface for a specific sensor. It is responsible for handling user interactions and displaying the sensor signal. When a user interacts with a sensor view, (opening a contact switch) the sensor view creates a new sensor signal and notifies its controller.

- **Sensor Controller.** The sensor controller is the core component of the simulator. It serves the following four purposes: i) distributing sensor signals from the view to signal handlers, ii) applying different *Signal Filters* to manipulate the sensor signal (adding noise), iii) simulating corrupted sensors by using *Manipulators* for generating faked signals, and iv) visualizing recorded or real-time sensor data which is fed into the simulator.

- **Signal Handler.** Components can register as signal handler for certain sensor signals. Our current implementation provides several pre-defined signal handlers which can be used for logging signals into files, to the network or to the console.

- **Signal Parser.** Signal parsers are needed when recorded sensor signals should be visualized. For each incoming sensor signal a corresponding parser must be registered at the input signal parser, otherwise the incoming signal will be ignored. The registration of the parsers is usually done by the environment.

Fig 1 shows the main components of the simulator and outlines their interaction among each other. The main purpose of the simulator is to generate test data. For this purpose the user can interact with the graphical sensor items (0) on the user interface (clicking on the sensor or sensed area). This interaction is captured by a *Sensor View* and triggers the generation of a new *Sensor Signal* (1). The *Sensor Controller* is notified (2) about the new signal and starts processing (3) it. During the processing step the sensor
controller applies available **Signal Filters** on the signal and notifies registered **Signal Handlers** (4) which coordinate a further processing (5) of the signal (logging into a file). If a **Manipulator** is enabled for a controller it behaves like a sensor view and randomly generates faked sensor signals (6). If the simulator is enabled for visualizing, it captures sensor input from the network or from a file and passes it to the **Input Signal Parser** (7). The input signal parser searches for registered **Signal Parsers** (8) to parse the sensor signal and directs the signal to the sensor controller (9) which triggers its visualization (A) on the graphical user interface.

![Diagram of Simulator Component Architecture and Interaction](image)

**Fig 1.** Simulator component architecture and interaction

### 5. Implementation and Evaluation Prototype

We decided to use Microsoft’s .NET Framework for implementing the simulator for the following reasons: *i)* the .NET Framework offers a sophisticated UI design support and allows rapid prototyping, *ii)* it provides a component concept for writing custom graphical controls [9], and *iii)* custom components can easily be integrated into the Visual Studio development environment. We bypassed the need to implement a graphical modelling framework ourselves, by using Visual Studio as graphical modelling environment. Therefore, we had to provide: *i)* custom written sensor controls, *ii)* a core simulation library, and *iii)* the basic environment application. This allows to simply drag and drop additional sensors (custom controls) on the floor plan layout contained in the environment application.

The main architecture and the interaction between the basic components have already been described on a high level in section 4. In this section we take a more detailed look at the implementation and applied design patterns, the benefits and disadvantages of the custom control concept, necessary steps to extend the simulator and problems we had to cope with. **Fig 2** shows a rudimentary class diagram of the current simulator implementation.
To decouple dependency relationships and facilitate the exchange of implementation classes we followed the paradigm of ‘programming to an interface and not to an implementation’ [10]. This allowed us to write one Sensor Controller implementation which can be applied for different types of sensors. Our first approach was to implement the controller based on generics, which would have offered the advantage to access class specific methods of a Sensor View or Signal without casting and provided to check for type compatibility during compile time. Unfortunately, the Visual Studio designer cannot render generic controls properly. Therefore, we had to use a more general approach and decouple relations among components via interfaces.

When developing the different sensor views our intention was to factorize commodities into a common abstract base class as we did for the model (AbstractSensorSignal). The .NET component concept theoretically allows to program custom components which derive from an abstract control, but the Visual Studio designer cannot render them properly. Therefore, we decided to introduce a concrete class (BaseSensorControl) which must not be instantiated, but contains common functionality for all sensor views (delegates to notify the controller, methods for updating the view). For modifying the signal data we introduced the pipes and filter pattern [8]. It allows defining one or multiple filters which are applied to the signal, before the controller distributes it to registered handlers. The filters can be seen as mathematical function composition which is applied on the signal in the registration order of the filter. In our implementation the controller always stores the original perceived signal internally, and performs all filter operations on copies of the signal (side effect free interaction). To fulfill the requirement for producing corrupted sensor signals we introduced an interface (ICtrlManipulation) which allows modifying the controller behaviour (sending sensor signals without user interaction).

One of our key requirements was to develop an extensible simulation environment which facilitates adding new sensors. To create a new sensor component a developer has to perform at least the first of the following four steps: i) provide a data model for the sensor (implement ISignal) and provide a custom control for rendering the data model (implement ISensorView or derive from BaseSensorControl), ii) provide an implementation for parsing serialized signal data to enable the visualization functionality for the new sensor (implement ISignalParser), and iii) optionally write a customized filter or manipulator for the new sensor.

Currently we have implemented an evaluation prototype of the simulator which provides full support (simulation and visualization) for simple sensors (binary switch, contact switch and temperature sensor) and complex sensors (motion sensor and pressure pad). Simple sensors only send a signal on activation whereas complex sensors periodically send signals after activation. Fig 3 shows a screenshot of our current prototype. For demonstration purpose we have equipped the floor plan of a flat with different sensors and simulated typical activities, like cooking, personal hygiene, etc.
The simulated sensor data can actually be logged into a file or be sent via the network to a server. For the motion sensor we have implemented a noise filter to blur the sensed position. Optional a manipulator which repeats the last sensed signal can be enabled for each sensor on demand. Currently only logged sensor data can be visualized by the simulator at real time. Therefore, we plan to extend the visualization feature and implement common visualization mechanisms, like fast forward, slow motion and sensor interaction highlighting. In addition we want to introduce a network interface which allows visualizing incoming data from a real life ambient system.

6. Conclusion

In this paper we argued that gathering test data for ambient intelligence environments is critical, due to several issues and simulation could tackle some of them. Our simulation approach features several benefits such as: i) hardware independent development, ii) objective comparison of different reasoning and classifying approaches, iii) cheaper production of test data than the installation of laboratories, and iv) evaluating cost and benefit of different sensing equipment. For producing a feasible set of test data a simulation system which supports heterogeneous sensor environments and allows the simulation of corrupted sensor data is required. Therefore, we presented a hardware and platform independent simulation environment which can either be used for generating or visualizing sensor data. Nevertheless, a simulation approach can never replace hardware-related tests and systematic evaluations under real life conditions and can therefore be seen as complement.

There are several application areas for our simulation environment. During the development phase the simulator can be used to produce test data and during the testing phase it can be used to visualize perceived data. In contrast to other simulators, our simulator provides the possibility to apply user defined filters on simulated signals, as well as the possibility to simulate corrupted sensors. These features can be beneficial if a system is required to be robust against noise and has to feature a high fault tolerance, which is typical for real world deployments. When planning an ambient intelligence system the simulator can be used to evaluate a suitable set of sensors and classification mechanisms. This allows defining and verifying quality attributes for ambient intelligence systems (creating system benchmarks, for example). After the deployment of the system, this simulator feature can be used in addition to analyze occurring
problems, like insufficient detection rates, unconsidered situations or corrupted sensors. This could be beneficial for maintaining and adapting already installed systems.

Ambient intelligence systems require a high level of customization which can significantly prolong the installation duration of such a system. Some systems solve this problem by providing default user profiles which are finally customized for the individual user after the installation. Using a simulator to produce such customized profiles in advance, before the deployment would be a considerable thought.

We developed our simulator with the objective to provide a prototyping tool for software engineers of ambient intelligence systems. The first implementation of our simulator is a prototype and sticks to the ‘form follows function’ principle. The practical application in our projects will show if the current design provides a feasible usability or if further usability studies are needed.

Acknowledgements

This research is partially funded by the Austrian Research Promotion Agency (FFG), the European Regional Development Fund (ERDF) in cooperation with the Upper Austrian state government (REGIO 13, Innovative Upper Austria, Health Cluster). Any opinions, findings and conclusions or recommendations in this paper are those of the authors and do not necessarily represent the views of the research sponsors.

References