A Tool Chain for a Lightweight, Robust and Uncertainty-based Context Classification System (CCS)

Henning Günther, Firas El Simrany, Martin Berchtold and Michael Beigl
Institut für Betriebssysteme und Rechnerverbund, Technische Universität Braunschweig

Abstract

In this paper we present a tool chain developed to support a Context Classification System (CCS). The CCS is especially designed to run even on lightweight commodity phones with a high detection rate and low calculation effort. The main design aspect considered while building the tool chain was the support for most common users and not only developers. Moreover, the CCS we are using proves to be a robust context recognition method, which supports the gain of both context classes and a fuzzy uncertainty value describing the confidence of their classification.

1 Introduction

Context-awareness has been a research topic for many years now. There has been a lot of progress since first appliances [1], "smart" environments [2] and very rudimentary "smart" devices [3], compared to nowadays context-aware systems [4] and activity recognition [5]. This progress was facilitated through a growing number of toolkits to visualize, transform, interpret and/or classify sensor data.

The Common Sense ToolKit (CSTK) [6][7] was an early toolkit which was mainly focusing on the data transformation and visualization. The CSTK is a collection of tools, written mostly in C++, that assist in the communication, abstraction, visualisation, and processing of sensor data. CSTK's core qualities are its real- time facilities and embedded systems-friendly implementation, providing ready-to-use modules for the prototyping and construction of sensor-based applications. Although it can be used for offline analysis (i.e., using recorded data files), CSTK is mainly envisioned as a tool to be applied in online fashion (i.e., using the sensor data as it comes streaming in).

A well known and also a very early toolkit for processing sensor data is the Context Toolkit [8]. The aim of this toolkit is to make it easy to build context-aware applications. They claim that context is difficult to use because, unlike other forms of user input, there is no common, reusable way to handle it. We are disagreeing with this proposition, because our approach presents mechanisms to build context classifiers which are reusable due to their simple and flexible design.

In [9] a GUI-based C++ toolbox is presented that allows the building of distributed, multi-modal context recognition systems by plugging together reusable, parameterizable components. The goals of this toolbox are to simplify the steps from prototypes to online implementations on low-power mobile devices, facilitate portability between platforms and foster easy adaptation and extensibility. Our approach differs in the level the online system gets distributed. We try to do the classifications on the device and then infer further context knowledge in a distributed way, whereas in this paper, we do not present a reasoning. The reasoning is part of our future work.

Compared to the named toolkits, this work is focusing only on one strain of sensor data processing, which starts at the data collection, over to the annotation, the system identification and ends in a Context Classifying System (CCS) that runs even on lightweight commodity phones as the OpenMoko platform. Also, the focus of the proposed toolkit is not the diversity of algorithms, but the simplicity of identifying a CCS. The diversity that can be covered though, lies in the platforms which are supported by the CCS.

The paper is structured as follows: In Sect. 2, we introduce the tool chain with it's different components from data collection to CCS. In sect. 3. the Context Classification System CCS is introduced. Sect. 4 is about the process of training context classifiers out of real life data using the tool chain and the CCS. The description of the Data Collection Tool (DCT) can be found in sect. 5 followed by the description of the Context Annotator Tool (CAT) in sect. 6. In sect. 7 we present th performance evaluations of the CCS, whereas between different implementations is compared. In the last section we conclude and show some future activities.

2 The Tool Chain

The tool chain consists of several tools, algorithms, interfaces and databases, all for one purpose, to most simply as possible identify a CCS. All links of the tool chain and their interconnections are displayed in figure 1.

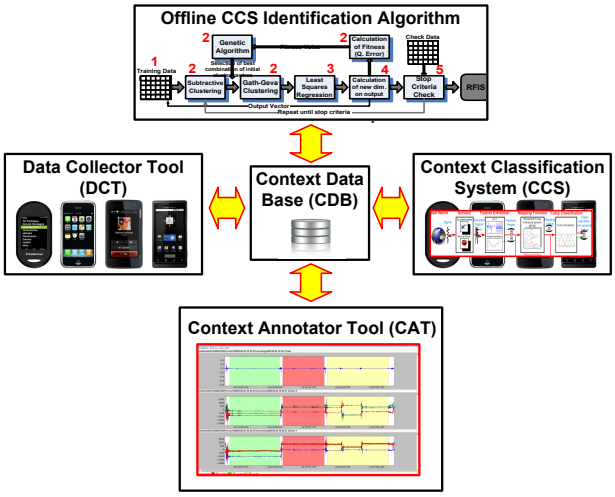

Figure 1: Tool Chain

2.1 Context Classification System (CCS)

Identifying the Context Classification System (CCS) is the sole purpose of the proposed tool chain. We chose a Recursive Fuzzy Inference System (RFIS) as the the mapping function from the features onto a classifiable linear set. Its accuracy compared to other methods (HMM, GMM, NN, etc.) is really high, it delivers a fuzzy uncertainty value with every classification and the calculation time rises linear with every new rule added to the RFIS.
The decision to use a RFIS mapping function in the CCS is founded, but there are other methods of mapping which are also suited. We decided to solely use the RFIS mapping, because a choice of different mapping methods would only confuse the user. So, having lots of choice in the mapping methods contradicts our conception of a simple to use tool chain and CCS, especially if the user is no expert.

2.2 CCS Identification Algorithm

The CCS identification algorithm consists of a combination of clustering, least squares and genetic algorithm. According to a training data set, a combination of subtractive and Gath-Geva clustering identifies the rules of the RFIS mapping function used in the CCS. The consequences of the rules are calculated with a least squares method. A genetic algorithm helps optimizing the cluster selection used in the RFIS mapping.
Here, we decided to develop our own identification algorithm, instead of using a well known approach such as the ANFIS (Adaptive Network-based Fuzzy Inference System) [10] one. One main aspect is, that the calculation time for a hybrid training (gradient decent) of the ANFIS is much higher than for our algorithm. Secondly, there is no easy way to include recurrence in the ANFIS approach, whereas the recurrence in our algorithm is a simple shift operation. Also, the accuracy of a trained multivariate ANFIS is lower than our used covariate RFIS. Since we use highly correlated sensor data (accelerometer axis), we need covariant cluster shapes, which are not supported in the ANFIS approach.

2.3 Data Collector Tool (DCT)

In order to get different contextual training data for the CCS (section 3) a simple Python application, the Data Collector Tool (DCT), allows to gather sensor data from the sensors on the Neo Freerunner (future support for other platforms, e.g. iPhone, Android, Maemo, etc.) which is equipped with one audio and two acceleration sensors. The DCT would be extended to gather data from other sensory inputs when needed. However the data produced should be first processed with the CAT, see section 4.1 before it becomes suitable for training. More on the DCT in section 5.

2.4 Context Annotator Tool (CAT)

The next step is the contextual annotation, and conversion into training data, of the collected raw data. The Context Annotation Tool (CAT), is a user friendly tool allowing visualization of different data sources and to easily associate suitable recognized context information. Then converting it into the training data, in a format suitable for the CCS. More on the CAT is shown in section 6.

2.5 Context Data Base (CDB)

All previously intoduced tools produce, consume or change data and classifiers. To manage the data, usually a stream pipes the data from one modul to the next one. Since we not only have one device producing sensor data or running a CCS, we need a more flexible approach to manage the data. We decided to use a data base, to store the raw sensor data, the annotated data, the preprocessed data and the trained classifiers.

The Contex Data Bases (CDB) purpose is to collect data from different users and devices (OpenMoko, iPhone, Motorola Milstone, Nokia N900, etc.) over a indefinite span of time. Also, all the trained CCS are not only stored, but also the training and check data are linked through a SQL string. With this SQL string the identification of each CCS can be reconstructed and therefore providing total visibility. The access to the CDB is done via command line or the Context Web Interface (CWI). Also, the CAT will be extended to allow direct loading and saving from and to the database. The implementation of the CDB is currently in progress, thats why we do not take a closer look into this module.

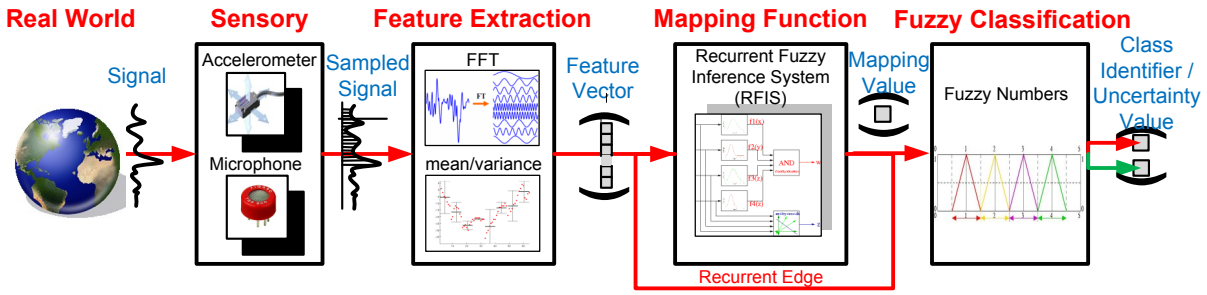

Figure 2: Online system architecture for classification and fuzzy uncertainty.

3 Context Classification System (CCS)

In our implementation we use the Context Classification System (CCS) for context recognition, which was introduced in [11] (there its called ORFC). The CCS uses a Recurrent Fuzzy Inference System (RFIS) for mapping the feature vectors on a classifiable linear set. All steps are diagrammed in figure 2.

3.1 Feature Extraction

For the application in this paper, we used audio and acceleration sensors. We segment the sensor streams into frames of 64 milliseconds length for audio and 80 milliseconds for acceleration. This frame lengths result in 512 samples (8kHz sampling rate) for audio and 8 samples (∼100Hz sampling rate) for acceleration. The features used for activity recognition with acceleration measurements are mostly variance and mean values, since they can be calculated with low resource consumption and give good classification results. These features were used to pre-process the accelerometer data, where the two 3-axis accelerometer sensors lead to a twelve-dimensional feature vector $\overrightarrow{\mathbf{v}}_t^{acc} = (v_1, .., v_{12})$. For audio data, we use " Fast Fourier Transformation (FFT)" to extract frequency features for the audio classification. Since the dimensionality after the FFT (e.g. 512 spectra) is too high to be used as input for a classifier, mean, variance and frequency centroid are calculated over the two halves of the frequency spectrum. This leads to a six dimensional feature vector $\overrightarrow{\mathbf{v}}_t^{snd} = (v_1, .., v_6)$.

3.2 Recurrent FIS Mapping

Takagi, Sugeno and Kang [12][13] (TSK) fuzzy inference systems are fuzzy rule-based structures, which are especially suited for automated construction. The TSK-FIS also maps unknown data to zero, making it especially suitable for partially incomplete training sets. In TSK-FIS the consequence of the implication is not a functional membership to a fuzzy set but a constant or linear function. The consequence of the rule j depends on the input of the FIS:

$$\begin{aligned} f_j(\overrightarrow{\mathbf{v}}_t) &:= a_{1j}v_1 + ..a_{nj}v_n + a_{(n+1)j} \\ &= \sum_{i=1}^{n} a_{ij}v_i + a_{(n+1)j} \end{aligned}$$

The linguistic equivalent of a rule with Gaussian membership functions in the antecedence part is formulated accordingly:

IF $\mu_{1j}(v_1)$ AND $\mu_{2j}(v_2)$ AND .. AND $\mu_{nj}(v_n)$ THEN $f_j(\overrightarrow{\mathbf{v}}_t)$

Since in activity recognition we deal with highly correlated features, especially when accelerometers are used, we use a slight variation of the rule above, which includes covariant Gaussian functions. The linguistic rule with a covariant antecedence is formulated accordingly:

$$\text{IF } \mu_j(\overrightarrow{\mathbf{v}}_t) \text{ THEN } f_j(\overrightarrow{\mathbf{v}}_t) \tag{1}$$

The covariant Gaussian MF, depending on the covariance matrix Σ_j and the mean vector $\overrightarrow{\mathbf{m}}_j$, is defined accordingly:

$$\mu_j(\overrightarrow{\mathbf{v}}_t) := e^{-\frac{1}{2}(\overrightarrow{\mathbf{v}}_t - \overrightarrow{\mathbf{m}}_j)\Sigma_j^{-1}(\overrightarrow{\mathbf{v}}_t - \overrightarrow{\mathbf{m}}_j)^T} \tag{2}$$

The whole antecedent part of each rule was multiplied with the usual TSK-FIS to get the respective weight, but with the covariant MF's the function is already the weight. The resulting formula for the covariant TSK-FIS is defined, as follows:

$$\mathbf{S}(\overrightarrow{\mathbf{v}}_t) := \frac{\sum_{j=1}^{m} \mu_j(\overrightarrow{\mathbf{v}}_t) f_j(\overrightarrow{\mathbf{v}}_t)}{\sum_{j=1}^{m} \mu_j(\overrightarrow{\mathbf{v}}_t)} \tag{3}$$

The outcome of the mapping at time t is fed back as additional input dimension for the TSK-FIS mapping at $t+1$. The recurrence not only delivers the desired uncertainty level, but also stabilizes and improves the mapping accuracy.

3.3 Fuzzy Classification

The outcome of the TSK-FIS mapping needs to be assigned to one of the classes that the projection should result in. This assignment is done fuzzily, so the result is not only a class identifier, but also a membership, identifying the fuzzy uncertainty of the classification process. Each class

identifier is interpreted as a triangular shaped fuzzy number (fig. 3). The mean of the fuzzy number is the identifier itself, with the highest membership of one. An example for four classes is shown in Fig. 3. The crisp decision (which identifier is the mapping outcome) is carried out based on the highest degree of membership to one of the class identifiers. The overall output of the RFIS mapping is a fuzzy classification, that is, a pair $(C_{\mathcal{A}}, \mu_{\mathcal{A}})$ of a class identifier and a degree of membership.

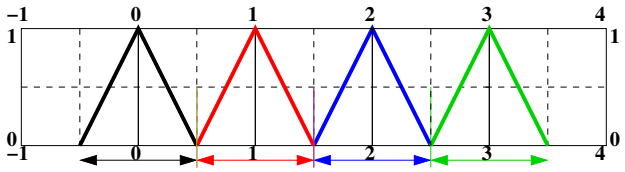

Figure 3: Fuzzy numbers identifying class membership and fuzzy uncertainty level.

3.4 Fuzzy Uncertainty Filter

The classifications vary strongly with respect to fuzziness and therefore in the reliability of the RFIS mapping. Since many more classifications are made than needed for most applications, a filter upon the fuzzy uncertainty can improve reliability, but also reduces the number of classifications. In order to determine the threshold values for filtering, we used the "'Receiver Operator Characteristic (ROC)" described in [14].

3.5 Implementation

The first implementation of the CCS was done on the Neo Freerunner phone, developed by the OpenMoko project [15].

3.5.1 Python

The Python implementation was done first to provide a proof-of-concept implementation to show that the system could deliver certain recognition levels. The implementation consist of three parts:

1. A rule parser
2. A rule engine
3. An input processing module

The rule parser uses the INI file format to load rules into the system. Each rule file starts with a *default* section in which the rule count as well as the dimensions of the rules are specified:

```
[DEFAULT]
dimensions = 13
rules = 4
```

Even though this information is redundant, it can be used for error checking or making the parsing process much easier. Following this header, each rule is given as a separate section. Each rule j is defined through the covariance matrix (*sigma* $\hat{=} \Sigma_j$), the mean vector (*mean* $\hat{=} \vec{\mathbf{m}}_j$), the consequence parameter vector (*consequence* $\hat{=} \vec{\mathbf{a}}_j = (a_{1j}, ..., a_{(n+1)j})$ and the bit masking vector (*bitvec* $\hat{=} \vec{\mathbf{b}}_j$ for example $\vec{\mathbf{b}}_j = (0000111010101)$):

```
[RULE1]
sigma = 2.502186e-02 -1.515808e-02 ...
mean = -8.953875e+02 -4.980525e+02 ...
consequence = 3.532613e-03 ...
bitvec = 0 0 0 0 1 1 1 0 1 0 1 0 1
```

With the bit masking vector $\vec{\mathbf{b}}_j$ the rule j of the CCS can be changed without destroying the original rule. The bit masking vector therefore gives the possibility to adapt the CCS to changed conditions or users. More details on the bit vector masking can be found in [16].
The rule engine essentially implements the algorithm described in section 3. All matrix-/vector-operations are implemented using the *numpy*-package [17], which provides optimized numerical functions. A class diagram of the rule engine can be seen in figure 4.

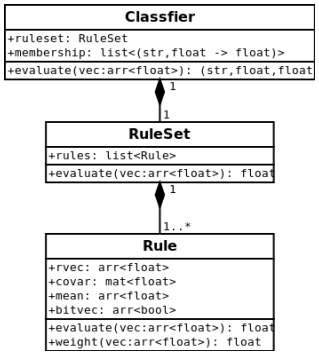

Figure 4: Class diagram of the python rule engine

The input processing module is responsible for fetching data from the sensor hardware of the phone and for preprocessing it according to section 3.1. Audio data is fetched using the ALSA (Advanced Linux Sound Architecture), which provides a clean API for setting the necessary parameters (sample rate, sample width, etc...). Preprocessing is done using *numpy* routines, except for the centroid calculation, which had to be implemented in python. Movement data is collected using the exposed *sysfs* API of the OpenMoko operating system.

3.5.2 Native (C)

The C implementation was conceived to address the main problem with the python implementation: performance. As we will show in section 7, the python implementation wasn't able to achieve real-time performance on the Freerunner phone.

To make the implementation as lightweight and portable as possible, as few as possible external libraries were used. The fourier transformation necessary for the audio preprocessing was implemented using the *FFTW*-library [18]. Everything else, including vector-/matrix-operations were hand-coded to avoid dependencies.

4 CCS Identification

In this section we show how to use the tools collecting training data and how the algorithm for identifying the CCS works upon the data.

4.1 Obtaining Training Data

Real life data collection, both from audio and acceleration sensors, was conceived in a way that, at the end of the tool chain, would produce a specific set of feature vectors as described in 3.1.
Contexts in real life are correlated, have intersections and can occur simultaneously. However we needed to define context classes which are simple and flexible enough to ensure re-usability and to avoid overlapping. E.g., it makes no sense to train a CCS (RFIS) with both acceleration and audio data. It leads to lower context recognition ratios and is semantically incorrect. Combining other kinds of sensor inputs, like video and audio or GPS and acceleration, would be more plausible. Consequent we choose to train contexts with one sensor type data at a time. Once the set of context classes, we wanted to collect data for, is defined, data is collected according to the following procedure:

1. Collecting raw data using Data Collection Tool DCT (section 5): Both from audio and acceleration sensors, using the DCT, we can collect annotated and non annotated data. Non annotated data is mere sensor data. Annotated data, is sensor data that was directly associated to a specific context class and then bundled into a tar file format. The tar file can be directly imported in the Context Annotation Tool CAT (section 6).

2. Processing the data using the CAT: Non annotated data will be annotated then converted into raw training data. Annotated data can be edited or directly converted.

3. Feature extraction according to section 3.1 produces ready to use training and check data.

All above described steps are visualized in figure 5.

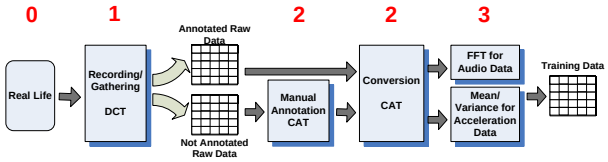

Figure 5: Training Data Gain using DCT, CAT and Feature Extraction

4.2 The Identification Algorithm

The mapping of sensor data features onto a classifiable set is done with a Recurrent Fuzzy Inference System (RFIS). The RFIS is identified upon an annotated training feature set via a combination of clustering algorithms, linear regression and genetic algorithm generalization. The algorithm for identifying the RFIS consists of five steps (Fig. 6).

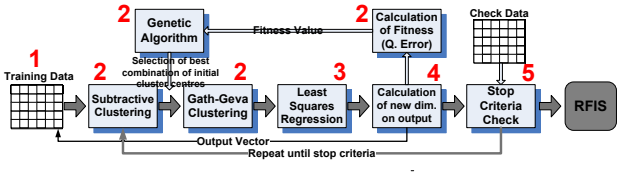

Figure 6: Offline CCS Identification Algorithm

1. Data Annotation and Separation: The training data are separated according to the class the data pairs belong to. Clustering on each subset delivers rules that can be assigned to each class. **2. Clustering:** Subtractive clustering [19] gives an upper bound on the amount of clusters. However, because it cannot adapt to covariant cluster shapes, it needs many fuzzy cluster functions to adapt to the data. A Gath-Geva clustering [20] is performed upon the set of cluster centers determined through the subtractive clustering. Since the number of clusters for the subtractive clustering is higher, a genetic algorithm searches for the best selection of initial cluster centers for the Gath-Geva clustering. **3. Least Squares:** A linear regression identifies the linear functional consequence of the rules. The least squares method minimizes the quadratic error, which is the quadratic distance between the desired output and the actual output of the TSK-FIS classifier for the training data set. Minimizing the quadratic error leads to an overdetermined linear equation to be solved. **4. Recurrent Data Set:** The recurrent TSK-FIS is obtained over a data set that has the output of the previously identified FIS shifted by one, so the first data pair of the training set has a zero in the recurrent dimension. All data pairs for time $t > 1$ have the output of the FIS mapping of $t+1$ in the recurrent dimension. For this data set the steps 1 to 3 are repeated. **5. Stop Criterion:** We could not find a general stop criterion, since two demands need to be met: the resulting classifier needs to have high accuracy and the outcome needs to have an uncertainty level that is profitable for reasoning. Therefore, the developer has to decide according to a separate check data set, which results are suitable for the classifier and its uncertainty levels. The steps 1 to 4 are repeated and graphically observed until these requirements are met.

4.3 Implementation

The CCS identification algorithm is currently implemented in Matlab. The implementation consists of different scripts for the clustering, the linear regression and the genetic algorithm. In future we want to translate the Matlab scripts into python, so not only users with a Matlab license can use the algorithm.

5 Data Collection Tool (DCT)

The Data Collection Tool is a simple Python application that gathers audio and acceleration data on the Freerunner phone. Upon startup, it fires up two threads: The audio thread records audio samples of the required frequency (8000Hz) and saves them as an audio file on the disk (either in WAV- or, to conserve space, FLAC-format). The recording time of the audio sample is given in the filename to ease the finding of previous recordings. The second thread reads acceleration data from the two acceleration sensors of the phone and writes them with their corresponding timestamp into a simple text file. The DCT also allows the user to associate contexts from a list of contexts to the gathered data. The user can whenever he wants change the current context annotation. The resulting gathered data and the the list of annotations is saved in the annotation package format, described in the following section 5.1.

5.1 Annotation Package Format

To provide easy data exchange between the tools that collect and manipulate context annotated sensor data, a new file format (fig. 7) was designed. The idea is to couple the annotation data with the actual sensor data it relates to.
An annotation package is a file that combines sensor data from multiple sources with annotation data. The outer layer of the file is a *tarball*-archive. It always contains an index file that specifies the content.

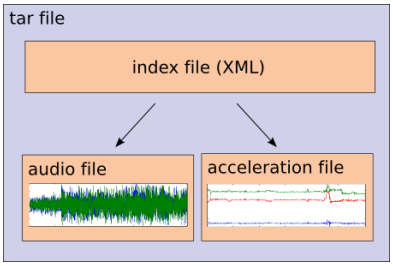

Figure 7: Annotation Package structure

Each sensor data is represented by a file in the tarball and an entry in the index file (fig. 8).The index file specifies meta-information for the sensor files, for example the start time of an audio file. Context annotations are also stored in the index file.

```
−<sensory-input>
  −<annotations>
      <annotation end="733663.512688" id="HGI" start="733663.512226"/>
      <annotation end="733663.513213" id="HGO" start="733663.512688"/>
      <annotation end="733663.513702" id="HGI" start="733663.513222"/>
      <annotation end="733663.514145" id="HGO" start="733663.513706"/>
      <annotation end="733663.514638" id="HGI" start="733663.514154"/>
      <annotation end="733663.515101" id="HGO" start="733663.514638"/>
      <annotation end="733663.515699" id="HGI" start="733663.51511"/>
    </annotations>
  −<sources>
      <movement file="mov_home0.log" name="mov_home0" sensor="0"/>
      <movement file="mov_home1.log" name="mov_home1" sensor="1"/>
      <audio channels="0,1" file="sound_home.flac" name="sound_home" offset="733663.512222"/>
    </sources>
  </sensory-input>
```

Figure 8: Example of an index file

6 Context Annotator Tool (CAT)

The Context Annotator Tool (CAT) is described in this section.

6.1 Purpose

To generate training data for the algorithm, the collected sensor data has to be annotated with the context in which it was collected. In order to associate sensor data with context information, it is necessary to

1. Read the sensor data from different formats
2. Display the sensor data
3. Allow the user to annotate the data with context information
4. Export the annotated data into a format suitable for the training algorithm

The Context Annotator Tool was designed to put all these mechanisms into one easy to use tool.

6.2 Design aspects

Most sensor data can be visualized as a two dimensional graph with a time and value axis. To be able to distinguish between different sensors, each sensor data should be displayed separately, but it should nonetheless be possible to see which sensor data happened at the same time. To achieve this, a horizontal layout, with the sensor data aligned to a global time axis, is a good choice.
Good visualization of the annotations can be achieved by overlaying the graph with colored boxes representing the contexts. To ease the process of associating a context to an audio sample, playback of audio samples was integrated by allowing the user to select a time slot and playing the contained audio data.

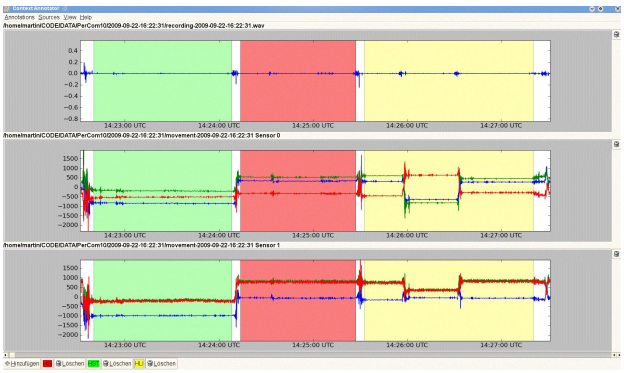

Figure 9: Screenshot of Context Annotator Tool (CAT)

6.3 Implementation

The Context Annotator Tool is written in Python (for class diagramm see fig. 10) to provide good extensibility, platform-independence and speed of development. For the data visualization the Matplotlib [21] toolkit was used, which provides Matlab-like plotting capabilities. The GUI is implemented on top of the GTK+ toolkit to allow the tool to run on as many platforms as possible. Audio loading, processing and playback is done using the GStreamer framework [22], which frees the implementation from worrying about things like audio codecs and output.

The central data structure of the tool is the *annotations* structure. It keeps track of all context annotations (start-/end-time and name). Other components can register as listeners if they want to be informed if an annotation is added, deleted or updated.

Each sensor data display is represented by a *display* component. It renders the sensor data using Matplotlib and reacts to user interaction (scrolling, adding new annotations, etc.).

The sensor data is managed by the *source* component. It is responsible for loading sensor data from various formats (audio file, accelerator log file, etc.).

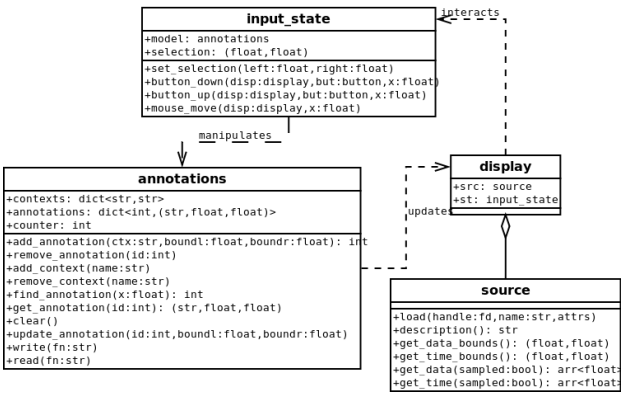

Figure 10: (Partial) class diagram of the CAT

The implementation is designed to be as flexible and extensible as possible: New sensor data types can easily be added to the system by simply extending the *source* base class. The class must implement methods to read and write the sensor data from or into a file, load and store the meta-information into XML and to query the user for extra informations (for example when the sensor data starts).

7 Performance Evaluation of CCS

The most critical performance criterion is the run time, since the algorithm has to be able to process data in real time. To evaluate the ability of the system to process the incoming sensor data fast enough to keep up, it is first necessary to find out how many classifications have to be done per second.

The audio system delivers 8000 samples per second; each classification requires 512 samples. Thus the system has to perform 15.625 audio classifications to keep up with the incoming data. Similarly, the acceleration sensor provides 100 samples per second and 8 samples are required per classification, so the system must be able to perform 12.5 acceleration classifications per second.

Each classification process can be broken down into two measurable units:

Preprocessing in which the data fetched from the sensor is transformed according to section 3.1.

Rule evaluation in which the feature vector is classified by evaluating the rules according to section 3.2.

The run time for each of these units for each sensor was taken by building the average over 1000 subsequent runs of the algorithm to account for fluctuations. Table 1 shows the running times for each implementation. It can be seen that the python implementation requires $0.006s + 0.040s = 0.046s$ per acceleration classification and $0.036s + 0.017s = 0.053s$ per audio classification. Because the system does 15.625 audio classifications per second, it requires $15.625 \cdot 0.053s = 0.828s$ processing time per second for audio classification and $0.575s$ for acceleration data, meaning that the system would require $1.4s$ of processing time per second, which means that the system would need 1.4 times the processing power it actually has to provide real time performance.

By significantly dropping the runtime of the accelerator preprocessing and the rule evaluations, the native C implementation can provide far better performance, leaving about half of the processing power of the CPU for other purposes:

	accel.		audio		req.
	preproc.	eval.	preproc.	eval.	
python	0.006s	0.040s	0.036s	0.017s	1.40
c	0.00012s	0.00093s	0.03255s	0.00047s	0.53

Table 1: Running times for different implementations

The numbers also reveal one of the weaknesses of the Openmoko hardware: It takes a long time to calculate the

fourier transformation, because the processor lacks a floating point unit. There could be an opportunity to improve performance by using an integer FFT. This possibility is however left open for future works.

8 Conclusion and Future Work

We have shown that it is possible to design and implement a tool chain that supports a Context Classification Systems (CCS), which is easy to use, transparent and robust. Also, the CCS can provide certain features, such as robustness, accuracy and uncertainty, other classifiers can not. Details of the general layout as well as implementation details were given for each tool involved in the chain. Two different implementations of the CCS were presented and real time performance of these implementations was evaluated. We were able to show that a real time classification system based on our approach is indeed not only feasible with current technology, but can also provide high accuracy and uncertainty in classifications.
As mentioned before, there are still many things to be done: We will proceed implementing the described Context Data Base and work on some performance issues of the Context Annotator Tool. As we designed the implementations to be platform independent, we also began porting the CCS to other platforms, for example the iPhone, the Motorola Milestone (Android) and Nokia N900 (Maemo5). This will allow more people to explore our tool chain and also extend our choices of sensors.

Acknowledgments: This work was funded by the NTH School for IT Ecosystems. NTH (Niedersaechsische Technische Hochschule) is a joint university consisting of Technische Universitaet Braunschweig, Technische Universitaet Clausthal, and Leibniz Universitaet Hannover.

References

[1] M. Beigl, H.-W. Gellersen, and A. Schmidt, "Mediacups: Experience with design and use of computer-augmented everyday artefacts," *Computer Networks, Special Issue on Pervasive Computing, Elsevier, Vol. 35, No. 4*, 2001.

[2] C. D. Kidd, R. J. Orr, G. D. Abowd, C. G. Atkeson, I. A. Essa, B. MacIntyre, E. Mynatt, T. E. Starner, and W. Newstetter, "The aware home: A living laboratory for ubiquitous computing research," in *Proceedings of the Second International Workshop on Cooperative Buildings (CoBuild'99)*, 1999.

[3] A. Schmidt, K. A. Aidoo, A. Takaluoma, U. Tuomela, K. V. Laerhoven, and W. V. de Velde, "Advanced interaction in context," in *Handheld and Ubiquitous Computing, First International Symposium, HUC'99*, ser. LNCS, H.-W. Gellersen, Ed., 1999.

[4] M. Buettner, R. Prasad, M. Philipose, and D. Wetherall, "Recognizing daily activities with rfid-based sensors," in *UbiComp*, 2009, pp. 51–60.

[5] V. W. Zheng, D. H. Hu, and Q. Yang, "Cross-domain activity recognition," in *Ubicomp '09: Proceedings of the 11th international conference on Ubiquitous computing*. New York, NY, USA: ACM, 2009, pp. 61–70.

[6] K. van Laerhoven, M. Berchtold, and S. Reeves, "Common sense toolkit (cstk)," 2004, http://cstk.sourceforge.net/.

[7] K. V. Laerhoven, M. Berchtold, and H.-W. Gellersen, "Accessing and abstracting sensor data for pervasive prototyping and development," *In the Adjunct Proceedings of Pervasive*, 2005.

[8] A. K. Dey and G. D. Abowd, "The context toolkit: Aiding the development of context-aware applications," in *In the Workshop on Software Engineering for Wearable and Pervasive Computing*, 2000.

[9] D. Bannach, K. S. Kunze, P. Lukowicz, and O. Amft, "Distributed modular toolbox for multi-modal context recognition," in *ARCS*, 2006, pp. 99–113.

[10] J.-S. R. Jang, "Anfis: Adaptive-network-based fuzzy inference system," *IEEE Transactions on Systems, Man and Cybernetics, 1993, vol. 23 pp. 665-685*, 1993.

[11] M. Berchtold and M. Beigl, "Increased robustness in context detection and reasoning using uncertainty measures - concept and application," *Proceedings of the European Conference on Ambient Intelligence (AmI'09)*, 2009.

[12] T. Tagaki and M. Sugeno, "Fuzzy identification of systems and its application to modelling and control," *Syst., Man and Cybernetics*, 1985.

[13] M. Sugeno and G. Kang, "Structure identification of fuzzy model," *Fuzzy Sets and Systems*, 1988.

[14] M. Berchtold, C. Decker, T. Riedel, T. Zimmer, and M. Beigl, "Using a context quality measure for improving smart appliances," *IWSAWC*, 2007.

[15] "Openmoko." [Online]. Available: http://www.openmoko.com/

[16] M. Berchtold, T. Riedel, K. van Laerhoven, and C. Decker, "Gath-geva specification and genetic generalization of takagi-sugeno-kang fuzzy models," *SMC08*, October 12-15 2008. [Online]. Available: http://www.ibr.cs.tu-bs.de/users/berch/publications/SMC08.pdf

[17] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online]. Available: http://www.scipy.org/

[18] M. Frigo and S. G. Johnson, "The fastest Fourier transform in the West," Massachusetts Institute of Technology, Tech. Rep. MIT-LCS-TR-728, September 1997.

[19] S. Chiu, "Method and software for extracting fuzzy classification rules by subtractive clustering," *IEEE Control Systems Magazine*, pp. 461–465, 1996.

[20] I. Gath and A. B. Geva, "Unsupervised optimal fuzzy clustering," *IEEE Transactions on Pattern Analysis and Machine Intelligence, vol 11(7), pp 773-781*, 1989.

[21] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science and Engineering*, vol. 9, pp. 90–95, 2007.

[22] "GStreamer – open source multimedia framework." [Online]. Available: http://www.gstreamer.net/