A Comparison between Decision Trees and Markov Models to Support Proactive Interfaces

Joan De Boeck* Kristof Verpoorten† Kris Luyten‡ Karin Coninx§

Hasselt University, Expertise centre for Digital Media
and transnationale Universiteit Limburg
Wetenschapspark 2, B-3590 Diepenbeek, Belgium

ABSTRACT

During the past few years, personal portable computer systems such as PDAs or laptops are being used in different contexts such as in meetings, at the office, or at home. In the current era of multimodal interaction, each context may require other interaction strategies or system settings to allow the end-users to reach their envisioned goals. For instance, in a meeting room a user may want to use the projection equipment and disable the audio output for a presentation, while audio input and output may be important while in a teleconference. In present computer systems most changes have to be made manually and require explicit interaction with the system. The number of different devices used in such environments makes that this configuration step results in a high cognitive load and causes interrupts of the tasks being executed by the end-user. In this paper we present how proactive user interfaces may predict the next interface changes invoked by context switches or user actions. In particular, we will focus on two machine learning algorithms, decision trees and Markov models, that may support this proactive behaviour for multimodal user interfaces. Based on some simple but relevant scenarios, we compare the outcome of both implementations in order to decide which algorithm is most applicable in this context.

CR Categories: H.5.2 [User Interfaces]: Input devices and strategies—Interaction styles ;

Keywords: Keywords Here

1 INTRODUCTION

Consider an employee with a laptop. At the office the laptop is used with a dual monitor and the audio level soft. When a telephone call is received, the employee needs his agenda tool and todo-list. Alternatively, when in a meeting, the laptop is either used for a presentation or just for taking notes, hence the appropriate settings to support these tasks have to be configured.

From the example, it may be clear that portable computer devices, such as PDAs or laptops, are more and more being used in a variety of situations, connected with a variety of input and output devices. With the current generation of user interfaces, this versatility still requires a manual adjustment of the interface settings for each particular context, obviously resulting in a higher cognitive load and a disruption of the task being executed by the end-user.

In our work, we investigate how proactive interfaces can be used to predict the next feasible interface changes invoked by a context switch or a user action. Proactive user interfaces monitor the current context and suggest the next possible steps based upon the user's behaviour learned from previous examples. In particular, this paper will focus on two machine learning algorithms, decision trees and Markov models, that may support proactive behaviour of such a multimodal interface.

The remainder of this paper is structured as follows. In the next chapter, we will first discuss related work. Next, we define the scope of this work and discuss the implementation of the tested algorithms. Section 4 describes the scenarios that are used for the test. The results and behaviours of the algorithms are discussed in section 5. We end our contribution by formulating our conclusions and suggestions for future work.

2 RELATED WORK

Proactive interfaces can already be found in several prototype application. Because consumer electronics are becoming more and more complicated and a lot of users are unable to understand their full potential, Leberman created "Roadie" [5]; a prototype interface that will try to infer the users' goal by monitoring his or her actions. Using Roadie, users can select their goal from a list of suggestions. Next, planning and commonsense reasoning is used to explain how to reach the goal.

Alternatively, Dey et al [4] propose *a CAPpella*, which uses programming by demonstration to teach a context-aware application. This program uses context data, such as raw audio or video from the environment in which the relevant parts are (manually) indicated. After several iterations of recording and indicating the relevant parts, the application should be able to recognise a certain context (such as a meeting), and hence perform the relevant actions for that context (such as launching a notepad to make notes).

Buyn and Cheverst propose the utilisation of context history together with user modeling and machine learning techniques, using *decision trees*, to create proactive applications. In [2] they describe an experiment to examine the feasibility of their approach for supporting proactive adaptations in the context of an intelligent office environment.

Cook et al. developed a smart home that adapts itself to its inhabitants. The role of the prediction algorithms within the architecture is discussed in [3]. They use three different algorithms, one of which is a Markov model. The Markov model is generated from collected action sequences and used to predict the next action, given the current context. This was tested on synthetic data generated for 30 days, using separate scenarios for weekdays and weekends. The algorithm generated a 74% predictive accuracy on that data.

Petzold et al investigate the feasibility of "in-door next location prediction" using a sequence of previously visited locations [7]. They compare the efficiency of several prediction methods such as Bayesian networks, neural networks and Markov models. Markov models score fairly well with an average prediction accuracy of about 80%.

From the related work, it appears that decision trees and Markov

*e-mail: joan.deboeck@uhasselt.be
†e-mail:kristof.verpoorten@uhasselt.be
‡e-mail:kris.luyten@uhasselt.be
§e-mail:karin.coninx@uhasselt.be

models may both fit in our proactive interface, as both solutions have proven to provide adequate output with a reasonable amount of learning samples. Therefore, we investigate which of both may be most suitable in our application.

3 Proposed Approach

3.1 Scope

As mentioned in the introduction, this research focuses on the computer interface of a portable device. When a contextual change occurs (such as receiving a telephone call or entering a meeting), the interface calculates the user's possible goal and proactively suggests the next possible actions as a result of previously recorded and processed learning samples.

We compare two approaches: one implementation is based on a decision tree algorithm; the second is based upon a Markov Model. The details of both approaches are described in the next sections.

It may be clear that the applications for those interfaces and the amount of system settings to monitor are nearly unlimited. However, we have chosen to work with a limited but relevant set of system settings and locations to ensure a manageable degree of complexity. We have defined the following attributes and values: Some applications that can be opened or closed (such as outlook, internet explorer, word, ...), some values for the screen's backlight, audio level, display mode (presentation mode or not) and screen resolution. Besides this, we also foresee some possible locations where the interface may be used in a different manner (Office, Meeting Room and Home). It may be important to note that context attributes such as "location" may be handled slightly different by the system as they are "read only" because the system cannot change them. Hence, a suggestion such as "go to the meeting room", must not occur.

As we are not concerned with capturing the data itself in order to avoid the technical problems which rise in this domain, we use a "Wizard Of Oz" user interface[1] as is described in section 4.1. In this form-based interface, we manually input the user's actions according to three predefined scenarios, as is described in section 4.

3.2 Decision Tree Implementation

3.2.1 General Approach

In a first part of the implementation we use *decision trees* [6] to learn from the user's actions. A decision tree is an internal data structure consisting of a "root node", "nodes", "connections" (branches) and "leaf nodes" as shown in figure 1. A node represents an attribute of the environment (e.g. location, screen resolution, ...), and may have one or more branches. Each branch contains a possible value for the attribute of its parent (e.g. "home", "meeting room" or "office" for the node representing the attribute "location"). Finally, a node that does not have children is called a leaf node, and it contains the final action.

A decision tree is built from a list of training examples. Such a training example is a collection of the current state of the attributes at the time of an action, together with that action. In the tree, the action of the training example will be reflected by the leaf node, while the values of the attributes can be found following the path to that node. A training example also contains information about the relevance of each attribute for the resulting action. It is not easy for a computer to "understand" which attributes are more relevant for a certain action than others. Therefore, a pragmatical approach is suggested in our implementation: attributes that have been changed more recently have a higher relevance. The underlying thought is that the reason for the user's current action is probably more dependent on the attribute(s) that lastly changed.

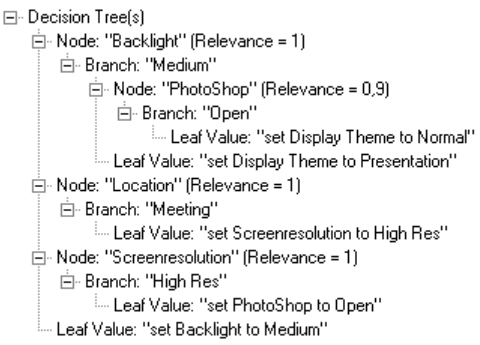

Figure 1: Example of a Decision Tree

3.2.2 Updating the Model

With each new training example, the entire tree is recalculated. In a first step, we have to decide which attribute is placed at the root. Obviously, this is the attribute that is relevant for most training examples. Therefore, among all training examples, the attribute with the highest relevance and the highest frequency is chosen. When a training example is found that is incompatible with the current root node (having other relevant attributes), it will create a new sub-tree with a new root, relevant for that particular example. This way, tasks that differ too much from each other are stored in different subtrees. For each branch the algorithm is recursively repeated: the most relevant attribute is chosen from the remaining training examples and is placed as a node in the particular subtree. When we get at the point where each of the remaining training examples contains the same action, this outcome is added as a leaf node. The result can be seen in figure 1. If more examples have a different outcome, but they cannot be further subdivided, the leave node contains all possible actions, together with their probability. Obviously, the more often a value occurs in the list of training examples, the higher its probability gets.

3.2.3 Making Predictions

To make a prediction, the decision tree is queried to find the next possible action that can be proposed to the user. Using the values we know for all attributes from the current application state, a suitable leaf node must be found. To find this leaf, we start at the root of the tree. At each node encountered, we follow the branch that has the same value for the given attribute as the value in the current application state. When a leaf node has been found, its value(s) is(are) returned as the result of the prediction. When no leaf node has been found, obviously there has never been such a learning sample, and hence there is no prediction.

It may also be possible to find a solution in more than one subtree. In that case, solutions are ordered according to the length of the path followed through the tree. The longer the path, the more attributes are successfully tested, and the more probable the outcome will be.

3.3 Markov Model Implementation

3.3.1 General Approach

In this part of the implementation, previously collected knowledge is stored in a first-order Markov model [1, 9], which may be seen as

[1] We use the term "Wizard Of Oz" here, because the application does not autonomously captures the context data, but instead the researcher has to manually input the data via a dialog-based interface.

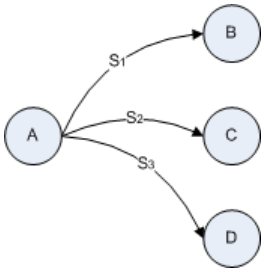

Figure 2: State transitions with their probabilities

a state transition graph where each state in the diagram corresponds to a state of the system, and each transition is annotated with its probability, as shown in figure 2.

In our implementation, we divide the list of attributes in two classes: first the attributes that may be changed by a suggestion of the system (such as open applications, audio level, etc.); secondly, we consider the attributes that are "read only", such as the location. Each attribute has a specific value, but additionally, when an attribute has not been changed since the startup of the system, its value is considered and stored in the model as "non relevant" (N/R). This is important for the generalisation of an observed sequence of actions, because it ignores irrelevant values.

When the system is started, it resides in a known state, having a value for each attribute. As none of the attributes have been changed yet, this initial state always maps to a state in the Markov model in which the value of each attribute is set to N/R. As soon as an attribute is changed, the new application state is compared with the internal Markov model. If there exists a state that exactly matches the current application state, this becomes the new active state of the model. If there is no match between the current application state and the internal Markov model, we are probably in a new learning path, and hence the new state is added to the model.

3.3.2 Updating the Model

While extending the model or navigating through it, the transition probabilities are continuously updated. In our implementation, we want recently occurred transitions to have a higher weight on the total probability, as they probably are more relevant in the learning process. A simple smoothing function, weighting past occurrences with a decreasing factor, is used as described in [8]. This function can be formulated as

$$\begin{pmatrix} S_1 \\ S_2 \\ \dots \\ S_n \end{pmatrix} = \alpha \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} + (1-\alpha) \begin{pmatrix} S'_1 \\ S'_2 \\ \dots \\ S'_n \end{pmatrix}$$

With S_i, the newly calculated probability and S'_i the current probability in the model. x_i is either 1 or 0 dependent whether the transition was chosen or not. α is a real number between 0 and 1 that indicates the weight of the current transition with respect to those in history.

3.3.3 Making Predictions

As soon as the user changes one of the system's attributes, the algorithm tries to predict the user's goal, and formulates a suggestion to assist the user to reach this goal. The suggestion can be derived from the Markov Model as follows:

First, all transitions starting from the current new state are analysed in a depth-first manner until all end-states are found.

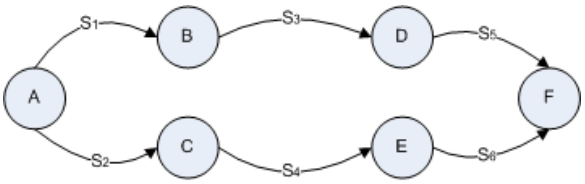

Figure 3: Multiple paths to the same target state

While searching for the end-states, the probability for each end-state is calculated. As shown in figure 3, multiple paths can lead to the same target. In that case, the probabilities are added up. For our example, this results in:

$$P_{a,f} = S_1 \cdot S_3 \cdot S_5 + S_2 \cdot S_4 \cdot S_6$$

If a loop (a transition to a state that already has been analysed) is detected, that path is ignored.

The result of the step described in this section is an ordered list of all end-states together with their probabilities, which may be *directly* reached from the current state.

A next step in our implementation is a comparison of the current application-state against the entire Markov model. This may find similar states from other recorded learning paths, resulting in a better generalisation of the learned samples. The "degree of similarity" is found as a result of a scoring system taking into account all matching, non-matching and non relevant attributes. This step provides us with a list of all states, ordered by their degree of similarity with the current state. Provided that their score is above a certain threshold, the n most similar states are selected and their respective end-states (with probabilities) are calculated in a similar way as described in the first step.

Finally, when the system has found all possible transitions to any relevant end-state with the respective probability, this ordered list is translated to concrete tasks that may assist the user, such as "Open a given application", "Set Audio Volume", ... It may be clear that end-states that suggest to adapt attributes that cannot be changed by the system (such as moving to another location) are ignored.

4 Comparison

As described in section 3.1, this research focuses on a user interface of a PDA or a laptop, that proactively suggests the next possible actions that the user may perform. This suggestion is triggered by any context switch, and derived from previously learned examples. In order to compare both implementations, we have conducted some experiments, each focusing on a specific scenario. In the next section, we first describe the "Wizard of Oz"-interface that will be used for the experiments. In the sections below, we then describe each scenario and its outcome. In order to draw our conclusions, for each experiment we count the number of user actions, as well as the useful, useless and wrong suggestions.

Although the amount of samples in each experiment is rather limited, it may give a good impression of the value of both algorithms. We believe more samples within the same scenario would lead to slightly different relevances or probabilities, but will not end up with significantly more complex models. It may be clear however that those experiments will not give a proof of the behaviour of the algorithms with prolonged use. Therefore, we refer to the "future work" section in this text.

4.1 Wizard of Oz Interface

In order not to cope with the "detection" of context or machine state and the integration in a "real" user interface, and in order to end

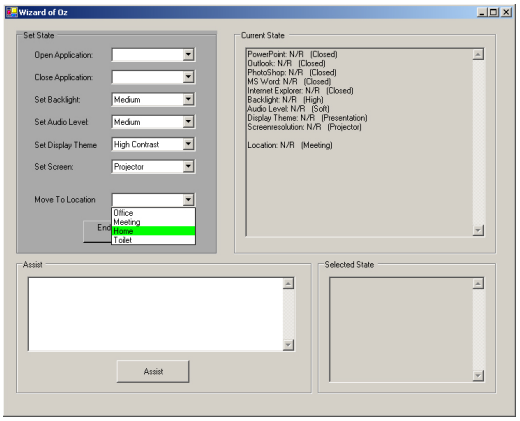

Figure 4: The "Wizard Of Oz" interface

up with manageable internal models, we have chosen for a limited set of attributes that are manually fed to the internal models via a simple dialog, as shown in figure 4.

It may be clear that this interface is only built for testing the underlying algorithms, and that ultimately the computer should be able to autonomously detect these (and other) system settings.

As soon as the algorithms are able to suggest some further actions, they are proposed in the bottom-most part of the window. As shown in figure 5, the listbox on the left gives the overview of all suggestions with their relevance. Relevances are calculated in percentages as a normalised result of internal calculations, but they must not be seen as an absolute chance. On the right, detailed information about the selected proposition is shown. Finally, by clicking the "assist" button, the selected suggestion is accepted and the corresponding application state is established.

4.2 Experiments and Observations

In the first scenario, we focus on the adaptations of the interface when a user moves to another location, independent of the machine's initial state. The experiment consisted of 5 subsequent trials, each starting from a slightly different initial computer state (initial screen resolution, initial audio level, initial location,...). In each trial the user moves to the meeting room where the audio level should be set to "off".

In total, this scenario required 9 actions for 5 trials. After the first trial in which, obviously, the audio level must be turned off explicitly, all sequent trials gave correct suggestions for both algorithms. Three times, a correct suggestion was proposed, and one time both algorithms should suggest to turn off the audio, but because the sound was already off in the initial state of that trial, the suggestion was suppressed.

The second experiment focuses on the different computer settings at different locations. For instance, at home, the audio should be "loud", the monitor is in "high resolution" and mostly the application "Internet Explorer" should be open, while "Outlook" is

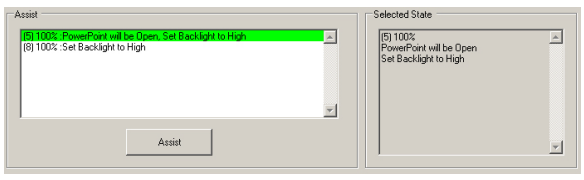

Figure 5: Example of a suggestion by the system

closed. On the contrary, at the office, the audio level is set to "low", there is a dual monitor setup and either "Outlook", "Word" or "Photoshop" are open.

The experiment consisted of 10 trials simulating to go home and to arrive at the office. In total, the scenario required 43 actions. Both algorithms gave 23 desired suggestions. The Markov implementation gave 3 unnecessary suggestions, while the decision tree implementation gave 4. Those suggestions where all the result of the fact that there is no application that is *always and consistently* opened at the office. Therefore, both algorithms may suggest to open applications that were recently opened, but are not necessary in the current sequence of actions.

In this experiment, we have to report that the decision tree suggested to turn on the sound as a next action after a suggestion to turn off the sound had been accepted. Because this can be easily solved by adding an extra check and this suggestion was at the very end of the scenario, when the desired state was reached, this was not recorded as a "false" suggestion.

In a third experiment we will compare how both algorithms may generalise learning paths from another situation, as well as how the algorithm behaves when exactly the same actions are performed in another order.

The first learning path contains changing the sound, theme and screen settings in an explicit location (e.g. meeting room). Next, the same sequence is desired, but now in an unspecified location. We see that the decision three has learned from the previous learning path, while the Markov model did not. This is because the decision tree is better at generalizing the learning samples. In a third sequence, we desire again a similar sequence of actions, but now at an explicit location, other than the first sequence (e.g. office). Now we observe that both algorithms have correctly learned from the second sample. For the decision tree, the second learning sample is preferred above the first, as it is more general.

Finally, we define a new learning path of four actions in a first sample, and do the same actions but in reverse order in a second sample. We see that after some actions, the generalisation extension of our Markov implementation finds the similarities and does a correct suggestion, while the decision three algorithm does not make any suggestions. It may be clear that, both algorithms will make correct suggestions when the second time the reverse order is presented, because meanwhile, both have learned from the user's actions.

5 Discussion

From the previous experiments, we can see that both implementations are behaving similar. In all scenarios the suggestions are nearly identical, and the number of unnecessary or false suggestions is low. We see that the decision tree implementation is better in generalising and applying learned data in another context, which could also be expected when reasoning about the algorithm. However, one can imagine that this may result in some general or false suggestions when using larger sample sets.

A Markov-model at the other hand, is better to recall the exact learned machine state, and may be less suitable for generalising the learned data. In this context, however, the addition to search for similar states may improve this drawback. Moreover, because of the latter addition, the Markov implementation is less dependent on the order in which actions are presented, as we found in experiment three.

Finally, extrapolating our results to a larger number of learning samples and more complex environments, both models may calculate suggestions that contain several actions to perform, or they may return a list of several possible final goals. As both numbers may grow too large, this can confuse the user. Therefore we suggest that the results are limited, both in terms of the number of suggestions,

as well as the number of actions within a suggestion. For instance, only the four or five most relevant suggestions, limited to four or five subsequent steps may be shown to the user.

Performance Considerations As the algorithms may run on a PDA with limited resources, some consideration about performance may be useful. It is clear that our test examples are far too simple to measure the performance, but based upon the description of the algorithms in chapter 3, we can make predictions about the time complexity. For both algorithms, we describe the time complexity when adding a new learning sample as well as when finding a suggestion. A thorough optimisation of the algorithms, however, may imply that some parts of the algorithms can result in another time complexity.

When using a *decision tree*, for each new sample, the decision tree must be rebuilt. For this purpose, all training examples have to be iterated once for each level in the tree. Searching in a well balanced tree is logarithmic with its number of nodes. With n the number of nodes and t the number of training samples, this results in $O(t \cdot log(n))$.

Searching for a single suggestion is performed by a single search operation in the tree, which is logarithmic with the depth of the tree. Multiple suggestions leading to a desired end-state are executed as multiple independent search operations within the tree. With l the length of the path to the end state, this gives $O(\mathrm{l} \cdot \log(\mathrm{n}))$. The value of l is typically small, as it has little sense to make tens of suggestions at once. This results in $O(\log(\mathrm{n}))$.

When using a *Markov model*, when an new sample is presented the entire model is analysed to find whether the new application state already exists in the model or not. This process is linear with the number of states m in the model. If necessary, a new state is created, and the probabilities of the transitions of the previous state are updated. As this is only a local adaptation, it is independent of the model's complexity: $O(\mathrm{m})$.
To make a suggestion, from the current state, all possible end-nodes are searched for, and the respectable probabilities are calculated. As loops in the graph are ignored, this is a linear function with the number of nodes present in the sub-graph (s). In a second step the Markov model is entirely analysed to find states that are "similar" to the current application state; this is an operation which is linear with the number of nodes in the graph. The time complexity is given by $O(s \cdot m)$. In a worst case scenario, however, s is equal to m (as it is a sub-graph), resulting in $O(m^2)$

As from our experiments, the complexity of the decision tree (n) and the Markov model (m) appear to be proportional, we can conclude from this reasoning that the decision tree implementation is a factor log(n) slower for adding new samples to the tree. Alternatively, the Markov implementation, also searching for similar states when making a suggestion, is slower (quadratic compared to logarithmic) for calculating a suggestion.

6 Conclusion and Future Work

In this work, we compared two algorithms that may be used to support a proactive user interface. As a result of a user action or a context switch, the interface proactively suggest the following probable actions. We described the implementation of both a decision tree algorithm and a Markov model. Both algorithms were tested using a "Wizard of Oz" interface which has been used to provide input according to some predefined scenarios.

From the results, we could conclude that both algorithms behave in a similar way, and that the differences are very small. However it appears that a decision tree is more suitable to generalise samples learned in a specific context, while a Markov model is more suitable when the exact learning sample must be recalled. Moreover a decision tree may sometimes generate false suggestions.

When considering the performance of both algorithms, we also see little difference. Although this is a preliminary conclusion, decision trees appear to be less optimal to integrate new samples in a large model, while our Markov implementation is worse in order to calculate a new suggestion.

As the experiments look promising for both algorithms, a more extensive evaluation may be conducted. Therefore, it is necessary to build an interface that is able to capture and change the relevant computer attributes and detect the current location or context. As this makes it possible to record the user's behaviour in practice, larger sample sets and more complex models can be built. Furthermore, a practical evaluation will also allow us to query end-users for their subjective satisfaction for either algorithm.

Acknowledgments

Part of the research at EDM is funded by EFRO (European Fund for Regional Development), the Flemish Government and the Flemish Interdisciplinary institute for Broadband Technology (IBBT). Funding for this research was also provided by the Fund For Scientific Research Flanders (F.W.O. Vlaanderen, project number G.0461.05).

References

[1] RD Boyle. Hidden markov models. `http://www.comp.leeds.ac.uk/roger/HiddenMarkovModels/html_dev/main.html`, 2005.

[2] H.E. Byun and K. Cheverst. Utilising context history to support proactive adaptation. In *Applied Artificial Intelligence, vol. 18, nr. 6*, pages 513–532, July 2004.

[3] D. Cook, M. Youngblood, E. Heierman, K. Gopalratnam, S. Rao, A. Litvin, and F. Khawaja. Mavhome: An agent-based smart home, 2003.

[4] Anind K. Dey, Raffay Hamid, Chris Beckmann, Ian Li, and Daniel Hsu. a cappella: Programming by demonstration of context-aware applications. In *Proceedings of CHI 2004*, Vienna, Austria, April 2004.

[5] Henry Lieberman and José Espinosa. A goal-oriented interface to consumer electronics using planning and commonsense reasoning. In *IUI '06: Proceedings of the 11th international conference on Intelligent user interfaces*, pages 226–233, New York, NY, USA, 2006. ACM Press.

[6] Tom Mitchell. *Machine Learning*. McGraw-Hill Education (ISE Editions), October 1997.

[7] Jan Petzold, Faruk Bagci, Wolfgang Trumler, and Theo Ungerer. Next location prediction within a smart office building. In *Proceedings of ECHISE 2005 - 1st International Workshop on Exploiting Context Histories in Smart Environments (held in Conjunction with the Pervasive 2005 Conference)*, Munich, Germany, May 2005.

[8] Peter Rigole, Tim Clerckx, Yolande Berbers, and Karin Coninx. Task-driven automated component deployment for ambient intelligence environments. Submitted to the Elsevier Pervasive and Mobile Computing Journal (PMC), 2007.

[9] Bernd Souvignier. Wiskunde 2 voor kunstmatige intelligentie, deel iii. probabilistische modellen. `http://www.math.ru.nl/souvi/wiskunde2_05/les12.pdf`, 2005.