Automatic task model generation for Interface Agent development

Victoria Eyharabide and Analía Amandi

ISISTAN Research Institute.
Facultad de Ciencias Exactas, UNICEN.
Campus Universitario, (B7001BBO) Tandil,
Buenos Aires, Argentina
{veyharab, amandi}@exa.unicen.edu.ar

Abstract

To give the user proper assistance, an interface agent should be able to predict the user's intentions. Usually, they achieve that through the use of a detailed task model for a particular application domain. However, to develop such knowledge representations is a difficult and time-consuming activity. Generally, it is tedious and annoying for the agent developers to express such knowledge manually. Also, the necessity to have an expert that offers information on the application domain is a problematic restriction. Therefore, this paper will focus on the automatic task model generation without previous knowledge on the application domain.

Keywords: Interface agent, Task model, knowledge acquisition.

1. Introduction

Currently, computers have become a fundamental piece both in business and private life; as a result of this a growing number of users must interact with them to carry out their daily activities. That computational overwork implies an increasing interaction between users and computers. To provide active assistance to the user with computer-based tasks, "Interface agents" have been developed [Maes94]. These agents are software components that can act as a personal assistant. These systems have the ability to detect the interests, preferences and habits from the user to which they assist. The agent acquires its competence by learning from the user as well as from agents assisting other users [Armentano – Amandi03].

The interface agent needs to know what the user is doing to assist him/her in the context in which he/she is working. If the agent knows the actions that the user is performing, it can offer him/her appropriate and opportune suggestions. To achieve this, it needs to infer which is the user's intention while carrying out a certain activity. Therefore, to facilitate the detection of such intentions, it is of vital importance to know all the possible tasks that can be carried out on the application. To do that, the agent should be able to identify the specific domain characteristics. That's why a key step to generate an application independent interface agent is to develop a detailed task model for that particular domain.

The task model generation is a time-consuming and difficult task that seems to require a fairly advanced knowledge of AI representations for plans, goals, and recipes [Garland et al.00]. That is the reason why agent's developers generally find arduous and difficult to express such knowledge. On the other hand, to acquire a task model starting from a domain expert is a complex problem. Since to transfer specific domain information from an expert to a designer is a constant problem in knowledge acquisition. In addition, the interface agents are used in very dynamic contexts; as a result, the model should be adapted easily to the possible modifications that the application can suffer. Beside, due to the task model describes particular information of a single application, it is specific for that application. For that reason, a task model already learned, in general, it cannot be reused.

Those previous reasons are the outstanding motivations that lead to the necessity of a straightforward task model acquisition without a domain expert support. Therefore, this paper will focus on the automatic task model generation with no previous knowledge on the application domain.

The rest of this paper is organized as follows. Section 2 introduces the basic notions of a task model. Section 3 describes the proposed solution for automatic task model generation. Finally, in Section 4 we present the work related to this research, and in Section 5 we sum up the presented solution.

2. What is a task model?

A task model is a template to describe functions of an organization in terms of tasks [Duursma]. A task could be seen like a recipe that describes the way in which the user can reach a goal. Informally, a task description specifies the minimum quantity of information that is needed to carry out a particular task, and the conditions to make the execution of that task possible. All those tasks as a whole constitute a specification called ***task model***.

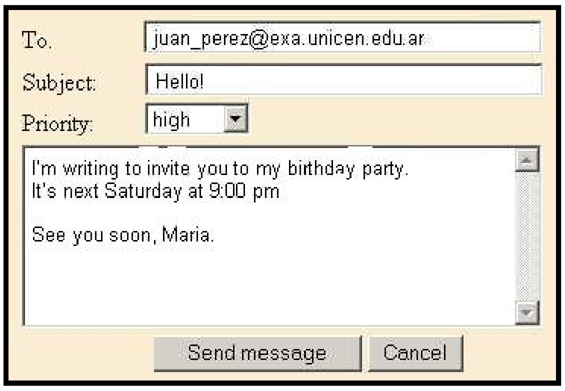
A task model is specific for a particular application. This representation responds to the question of what actions can be performed on the system and which information is needed to achieve them.

The great variety of techniques and notations developed at the moment [Bauer99], [Garland et al.00], [Garland02], [Paterno et al.97] show that currently there is no agreement on how a task should be adequately and completely described. However, there is a general agreement about the central role of the following information [Bomsdorf et al.98]:

- The structuring of tasks in subtasks.
- Pre and post conditions for tasks.
- Temporal relations between tasks.
- Objects used within a task context.

The best-known techniques for task modeling are based on hierarchical task representation, where tasks are decompose in subtasks and have an order of execution. That is to say, each recipe describes a set of steps that can be performed to achieve that abstract action [Garland et al.00]. In general, these recipes include preconditions, effects and subgoals that compose the execution of the task.

To exemplify the use of task models let us consider the case of an e-mail delivery. In that situation, the e-mail application introduces the user to a window in which, as minimum, he/she can complete the e-mail address, the subject and the text message, together with two buttons to accept or cancel the delivery. The figure 1 shows an example of that application.


To: juan_perez@exa.unicen.edu.ar
Subject: Hello!
Priority: high

I'm writing to invite you to my birthday party.
It's next Saturday at 9:00 pm

See you soon, Maria.

Figure 1: An e-mail application interface

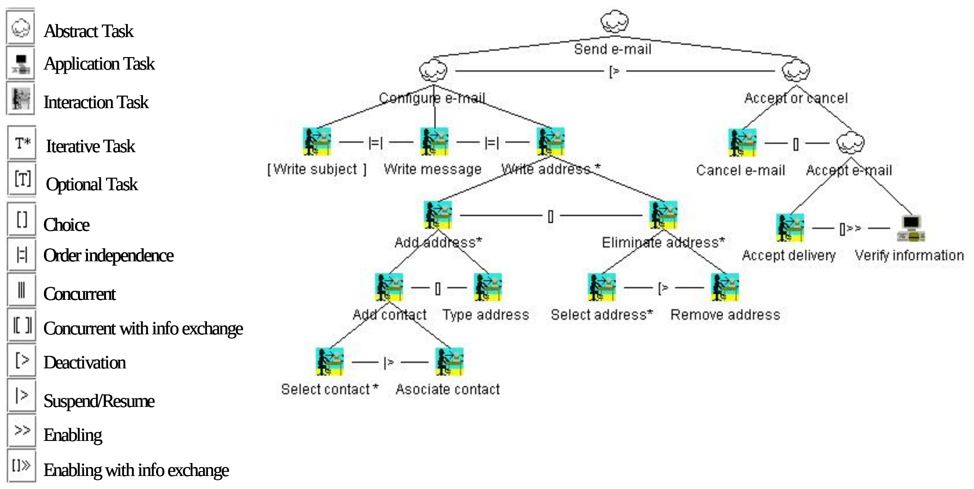


Figure 2: Task model for an e-mail delivery

Although, that functionality is presented on a single screen, to generate a task model that represents it, it is not a simple activity. Specially, it is necessary to incorporate high-level tasks to group related functionality and to eliminate ambiguities. For example, a possible task model is shown in figure 2 using the ConcurrTaskTree notation [Paterno et al.97].

There are two main abstract tasks called “Configure e-mail” and “Accept or cancel” related by the disabling connector ([>). This connector implies that once the “Accept or cancel” task has been executed; the task “Configure e-mail” cannot be executed again. To configure the e-mail, there three tasks that can be carried out in an independent way, that is to say, without a specific order (| = |). These tasks are “Write subject”, “Write message”, “Write address” and “Set priority”. The former and the latter task, which are between brackets ([]), are optional tasks. That means that it is possible to send an e-mail without a subject or without setting its priority. However, it is necessary to specify the text message and the e-mail address. Also, there can be iterative tasks like “Write address”, meaning that many addresses can be added.
Enabling with information exchange ([] >>) is another important relation between tasks. For example, this connector is between “Accept delivery” and “Verify information” symbolizing that after having accepted the e-mail delivery the information should be checked, and also, the former should give the data to the latter to verify it.

1.1. Advantages of a task model use in interface agents

The task model has become fundamental in the development of interface agents. Not only because it simplifies the detection of the user's intentions, but also, because the more knowledge about the user’s tasks is available, more it can be exploited when defining the system's properties and features, leading to a higher degree of user acceptance and user satisfaction [Bomsdorf et al.98].

Besides, the task model allows the agent to be detached from the application. That detachment enables their incorporation in existing applications without the necessity to modify them. Another advantage of that division is that it allows the agent to be independent from the application, a key requirement in the development of such systems. For example, let us consider the situation in which the application changes. It is the task model that absorbs these changes and the agent remains isolated from those modifications. Consequently, if the task model is the nexus between the application and the agent, it implies that it should absorb and reflect all the changes that take place in the application.

On the other hand, a hierarchical task model facilitates the high-level goal recognition because this representation decomposes the tasks in subtasks, allowing the detection of high level tasks (abstract tasks) while performing low level tasks (concrete tasks). In addition, this distribution allows the detection of dependent or independent conditions between the different tasks. Thus, to determine what tasks are related or not to the one that the user is developing is very important for the agent to offer suggestions related to the context in which the user is working.

1.2. Difficulties that developers find while building a task model

To build a task model is not a simple work. The fact of developing such knowledge representations is a significant challenge, since to specify them by hand is a difficult and time-consuming activity. This approach rise to the notorious knowledge acquisition bottleneck: developing an accurate domain model is a significant engineering obstacle [Garland et al.01].

Task model detection is problematic for several reasons. The first limitation is that agent designers have problems to identify and to understand specific domain tasks. These difficulties come up because, generally, designers are not specialists in the application field. On the other hand, domain experts typically are not agent designers [Cheng-Man et al.98]. Therefore, the necessity to have an expert that offers information on the application domain arises. This is a problematic restriction since that expert is not always available. More over when the agent is designed for applications previously developed. Also, although that specialist exists, it is generally tedious and annoying for people to express such knowledge manually.

However, a good task representation needs to contemplate all the essential conditions that an observed sequence of actions should satisfy to be recognized as belonging to that task, it should not be excessively restrictive. That is to say, it should not contain conditions that are not crucial to execute its associate task. The ideal task decomposition should be restrictive enough (to discriminate against the competitive hypotheses when comparing it) without limiting the possible involved behaviors.

Up to this moment, it has not been possible to reach an agreement on a standardized representation to describe a task model. In the analyzed works [Bauer99], [Bomsdorf et al.98], [Cheng-Man et al.98], [Garland et al.01], the notion that different task models can coexist for the same application arises with great emphasis. This is based on the fact that diverse recipes can describe a single task. Since recipe is defined as the group of steps that can be performed to achieve an action. Those steps can include different restrictions, impose diverse temporary orders between them, as well as dissimilar logical relations between their parameters.
Another inconvenience is the generally low quality in the knowledge elicitations with the user. Even more, sometimes it is difficult to obtain unambiguous data from a domain expert. Additionally, adaptation through the time is very important. It could be necessary to modify the models learned, even those ones broadly used. In that situation, it is important to verify the structure easily and to upgrade it in consequence.

Those inconveniences exposed previously, constitutes some of the reasons to generate the task model in an automatic way instead of constantly depend on a domain expert.

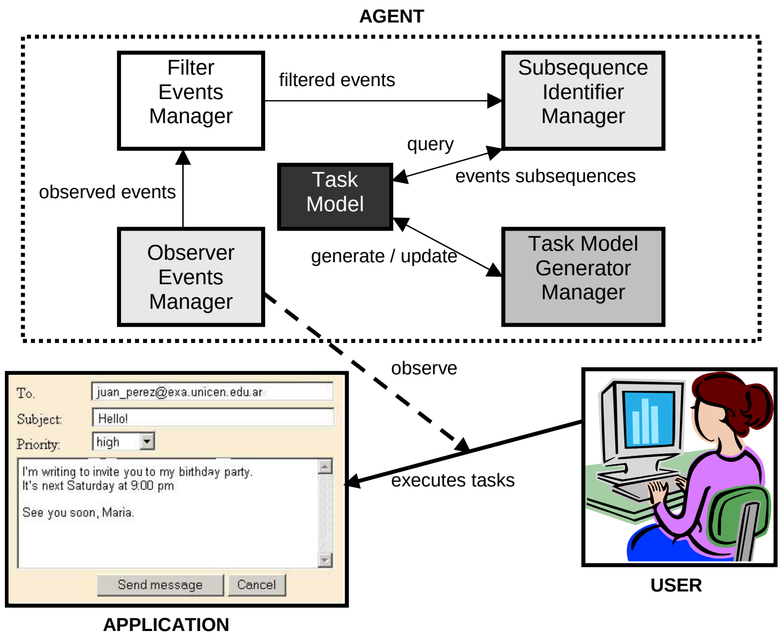
3. Building a task model automatically

As we have already mentioned before, a task can be seen as a recipe that describes the way in which the user can reach a goal.

In other words, a task execution has as final purpose the satisfaction of a user intention. The materialization of such tasks is simply achieved by the execution of some of the operations provided by the application. Although a task refers to those activities that can be performed within the application, for us it also includes smaller and more concrete actions, such as mouse click or a key pressed. Thus, the actions that the user carries out on an application are translated in events that are shot from the application interface. Summarizing, to satisfy his/her intention, the user performs tasks that become events inside the system when they are executed. Consequently, we could conclude that the user reaches his/her goals through the effects that events cause in the system.

Therefore, we have found a way to identify tasks automatically. If we capture the events that take place in the system, we can discover the operations executed by the user. And later on, starting from those operations we can identify the task associated to that group of actions.

With that purpose in mind, we have developed an algorithm to generate the task model automatically. That system observes the user while he/she is working on the application to capture the events. The principal components of the algorithm are presented in figure 3.



To: juan_perez@exa.unicen.edu.ar
Subject: Hello!
Priority: high

I'm writing to invite you to my birthday party.
It's next Saturday at 9:00 pm

See you soon, Maria.

Figure 3: Our system principal components

As is showed in the picture, the *Observer events manager* is the component in charge of the user observation. This manager main function is to capture all the events that happen inside the system while the user is working on the application. That sequence of events is given to the *Filter events manager*, which filters all the spurious events that are not significant to identify tasks. Therefore, the *Subsequence identifier manager* recognizes in that filtered sequence of events some possible subsequence that can make match with the tasks identified previously. Finally, those candidate subsequences are sent to the *task model* which updates itself with the aid of the *Task model generator manager*. All those principal components are clarified in the following sections.

1.3. Difficulties to overcome

Some obstacles make difficult the task recognition. One of those difficulties is the lack of knowledge on the application domain. At this point, it is important to emphasize that in this work there is not a previous knowledge on the application domain. This is an imposed characteristic to make the resulting system completely independent of the application.

On the other hand, many sequences of actions seem similar although they really correspond to totally different tasks. Another problem is due to the fact that in many situations users change the task they are performing and they begin to carry out another one without having concluded the previous one. So, the agent should discern whether the sudden change is due to a variation of the initial task, to an optional step that had not been carried out previously, or to a completely different task.

In spite of this, when the user works in the application, he/she performs tasks without making any indication of when a task finishes and the following one begins. Therefore, the events generated by a task are immediately after those events of the previous task and exactly before the events of the following one. Consequently, we obtain a continuous sequence of events. To exemplify this situation, let us consider three possible tasks, T1, T2 and T3. Also, to simplify the situation, we will represent the events with characters. So, the sequence of events [a, b, c] corresponds to the task T1, [p, q] to T2 and [i, j, k] to T3. If the user executes the tasks T3, T1 and T2 in that order, we could observe:

$$\underbrace{i,\ j,\ k,}_{T3}\ \underbrace{a,\ b,\ c,}_{T1}\ \underbrace{p,\ q}_{T2}$$

However, if the user doesn't generate any pause between the executions of the tasks, all the events appear with the same frequency one after another. Since we don't know the subsequence that corresponds to each task, we cannot know in advance neither the events, neither the limits of the executed tasks.

1.4. Listening the events

We should capture the events caused by the user's tasks. Those events could be a mouse click, a key pressed, a checkbox selection, etc. To visualize this idea more clearly, let us consider the example presented previously. In that case, the user has already completed all the corresponding fields and he has pressed the "Send message" button to complete the e-mail delivery. In consequence, the following events have been shot in the system:

1. The new e-mail window is activated.
2. The address textbox gets focus.
3. A key pressed for each character of the e-mail address.
4. The subject textbox gets focus.
5. A key pressed for each character of the e-mail subject.
6. The priority list gets focus.
7. The selection of the "high" item in the priority list.
8. A key pressed for each character of the text message in the e-mail text area.
9. A click on the "Send message" button.

The previous list corresponds to the sequence of events that are generated when the user sends a new message. Therefore, it is the succession of actions that match with the e-mail delivery task.

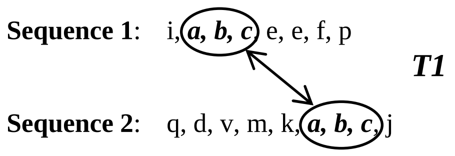
1.5. Spurious events filtering

The spurious events between the relevant ones are another obstacle that comes up. When we listen the system events, we do not only capture the events that correspond to specific tasks, we also obtain the events that are not associated to an application task. For example, we capture all the mouse movements, all the clicks that the user had made, all the components that get focus when a window is activated, etc. Therefore, the first challenge consists on filtering in the continuous sequence of events, which ones are not significant for the task recognition. Finally, we group all those events that are strongly coupled.
To exemplify this process, let us consider the case when the user writes a text. Each time the user presses a key, the system shots three different events; an event when the key is pressed, another one when the key is typed and finally a last one when the key is released. First, we eliminate two of them to obtain only one event per character typed; and at the end, we put together into one event all those character events that has occurred in the same text field. Thus, we summarize in a single event the text written by the user.

1.6. Task recognition

Although we cannot know in advance the subsequences that conform tasks, we can obtain different continuous sequences of events each time the user interact with the application. That is to say, whenever the user works on the application (since the execution of the application start until it finishes completely) we can obtain a different sequence of events. Then, to identify tasks we should discover repetitive behaviour patterns inside those different continuous sequences of events.

For example, let us suppose that the user has worked in the application twice, so we can obtain two different sequences of events: [i, a, b, c, e, e, f, p] and [q, d, v, m, k, a, b, c, j]. Our purpose is to find subsequences that are in both sequences. Then, as we can observe in the following figure, we could identify a task T1.


Sequence 1: i, ***a, b, c*** e, e, f, p
T1
Sequence 2: q, d, v, m, k, ***a, b, c,*** j

1.7. Task model representation

We represent the hierarchical task model as a binary tree. This representation contains information about the structuring of the tasks into subtasks and about the order of their execution. This is a simplified view upon the tasks which reflects only part of their multiple dependencies and the real task situation [Armentano – Amandi03].
The task model is composed by a set of hierarchical tasks. Each one of those tasks could be a concrete or a compose task. A compose task has subtask, whereas a concrete task is a specific low-level task which has its associated sequence of events. As we represent the task model as a binary tree, a compose task is composed by two task, a left and a right one.
In a hierarchical task model, there are different types of relations between the tasks. In this research, those relations are classified as binary or unary, depending on the number of the tasks involved in the relation. A binary relation is a connexion between two tasks, and a unary relation modifies only one task. Up to this moment, the relations provided are the following ones:

Binary relations

- **Order independence**: It is indispensable to execute both tasks although there is not a temporary restriction between them. When the execution of one task begins, the second one could not start its execution until the first one finishes.

- **Choice**: the execution of only one task is possible. If the first task was executed, the second one could not be performed and vice versa.
- **Concurrence**: In this situation the tasks can be executed concurrently. That is to say, any kind of temporary restriction exists between them.
- **Enabling**: The execution of one task enables the execution of the other one. After being executed one task, the execution of the other one could start.
- **Disabling**: The execution of one task disables the execution of the other one.

Unary relations

- **Iteration**: The task can be executed many times.
- **Optionality**: It is not necessary to execute the task.

1.8. Task model generation

The generating/updating process is as follows: The task model receives a possible sequence of events to be updated. The model asks its tasks which one is the most similar to the candidate sequence. To be compared each task has a similarity vector made from all the events that the task admits. The purpose of that vector is to make a quick comparison between two sequences of events. By doing this, we can discard an enormous quantity of tasks that are not similar to the candidate sequence, reducing the search space. With the tasks that pass this first check, there is a second and more deeply comparison to find the real similarity. Therefore, if the similarity between the candidate sequence and the most similar task is beyond a predefined external threshold, that task is updated with the sequence.

Once the task to be updated is found, the candidate sequence goes down through its subtasks repeating the previous process until the similarity between them it is under an internal threshold. In that moment, we have found the specific subtask that should be updated. At this point, the task model asks the Generator manager to update that particular task. The generator should update the relation between the existent tasks or create a new task if it is necessary.

1.9. Identification of relations between tasks

In this section, we are going to explain how to detect some of the principal relations between tasks supported by the task model.

Order independence

If two tasks T1 and T2 are linked with an order independence connector, both tasks should be executed, and also the task executed first should complete all its subtasks before the second one could start its execution.

To identify this relation, we should find in all the evidences that the execution of both tasks is always complete but a temporary order doesn't exist between them. That is to say, any task can begin its execution. Let's see an example:

If T1 = [a, b, c] and T2 = [p, q] and given the following sequences of events, where "x" means any event type:

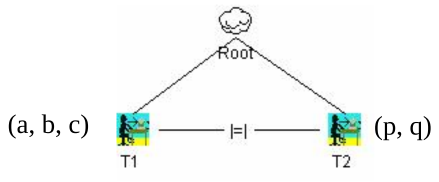
T1 ***T2***

Sequence 1: x, x, **a, b, c** x, x, x, x, **p, q**

Sequence 2: **p, q** x, x, **a, b, c** x, x, x, x

T2 ***T1***

Then, given the presented evidence, we can assure with a certain confidence level that the task T1 has an order independence relation with the task T2.



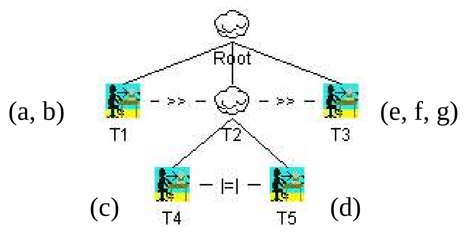
Another example in which we can detect this connector is presented next. Let us suppose the following two continuous sequences of events:

Sequence 1: x, x, **a, b, c, d, e, f, g**, x, x

Inverted

Sequence 2: x**, a, b, d, c, e, f, g**, x, x, x

In these sequences we can observe that although both subsequences seems similar, two events are inverted. However both events are necessary, this situation indicates an independent order in their execution. Thus, we can obtain the next task model:



Choice

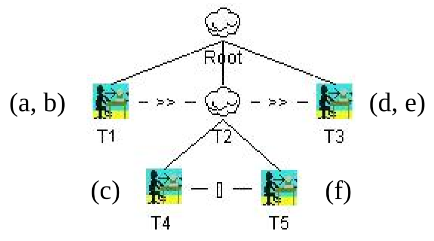
If two tasks are linked with a choice connector only one of them can be executed. To identify this relation, let us see a possible example:

Sequence 1: x, x, **a, b, c, d, e,** x, x, x, x

Different

Sequence 2: x**, a, b, f, d, e,** x, x, x, x, x

In the previous highlighted subsequences we can distinguish two different events. As a result, we can conclude with a certain confidence level that to perform that task the user must choose between the execution of the event c or the event f, but not both of them. Consequently, the resultant task model is:



4. Related work

The knowledge about what tasks the user can perform on an application brings countless utilities. For that reason, many fields of computer science—planning, intelligent tutoring, plan recognition, interface design, and decision theory to name a few — take advantages from applying general purpose algorithms to domain-specific task models [Garland et al.01].

Although task models are widely used, little effort has been made on how to generate them. Up to this moment, there are not too many works related to the way to generate a correct task model for a certain application. This problem is particularly difficult with little domain knowledge, a common situation, for example, when an agent is developed for an existing application.

In [Cheng-Man et al.98] Chung-Man Tam et al. implemented and evaluated an interactive tool called the User-Task Elicitation Tool (U-TEL). In order to guide the design of a user interface, the tool requires a domain expert to outline informal user-task models that an engineer can formalize using a model editor.

Brown et al. described in [Brown et al.] a utility theory-based approach to predict user intent by incorporating the ability to explicitly model users' goals. The main component of their architecture is a Bayesian network-based user model. They also capture environmental events to update that Bayesian network. However, for each observable event the host application should send an explicit message, via a message-passing interface, to the interface agent.

Some preliminary ideas are discussed by Garland et al. in [Garland et al.00] for making it easier to construct and evolve task models, either through interaction with a human domain expert, through machine learning, or in a mixed-initiative system.

On the other hand, Mathias Bauer in [Bauer99] presents an approach to the automatic acquisition of plan decomposition from sample action sequences but having a previous knowledge of the application domain. Alternatively, the task model development environment presented in [Garland et al.01] is centered on a machine-learning engine that infers task models from examples. The environment gives support for a domain expert to refine examples as he/she develops a clearer understanding of how to model the domain.

As we can see, the majority of the analyzed related works needs an expert domain support to model the user's interaction with the application. In contrast, we generate a task model neither having previous domain knowledge nor requiring an expert to offers information on the application domain.

5. Summary

We have presented a way to generate a task model automatically without having a previous knowledge in the application domain. To do that, we first observe the user's interaction with a specific application to capture the system events caused by the execution of the tasks performed by that user. So, we obtain a continuous sequence of events (since the user starts using the application until he/she finishes it) each time the user works on the application. Since we get all the events that occur inside the system, we should filter them to eliminate the ones that are not essential for the task recognition. Finally, we search for repetitive behavioral patterns inside those different continuous sequences of events to identify the task that the user has carried out. During that identification process we also look for variations in the execution order of those similar subsequences of events to identify the relations between the recognized tasks, as well as their special execution conditions.

References

[Armentano – Amandi03] Armentano, Marcelo and Amandi, Analía. ‘Agents Detecting User´s Intention’. In Proceedings of ASAI’03, Argentine Symposium on Artificial Intelligence, JAIIO'03. Bs. As. Argentine. (September 2003).

[Bauer99] Bauer, Mathias. ‘From Interaction Data to Plan Libraries: A Clustering Approach’. DFKI–Germany. In Proc. 16th Int. Joint Conf. on AI, 962–967. (1999)

[Bomsdorf et al.98] Bomsdorf, Birgit and Szwillus Gerd. ‘Coherent Modeling & Prototyping to Support Task-Based User Interface Design’. In Chi’98 Workshop, Los Angeles, California. (April 1998).

[Brown et al.] Brown, Scott; Santos, Eugene and Banks, Sheila. ‘Utility Theory-Based User Models for Intelligent Interface Agents’. Air Force Institute of Technology, USA – University of Connecticut, USA.

[Cheng-Man et al.98] Chung-Man, Tam R.; Maulsby, David and Puerta, Angel. ‘U-TEL: A Tool for Eliciting User Task Models from Domain Experts’. Stanford University, USA – Aurelium Inc., (Canada. 1998)

[Duursma] Duursma, Cuno. ‘Task Model definition and Task Analysis process’. Free University of Brussels. AI Laboratory. Pleinlaan 2. (B1050) Brussels – Belgium.

[Garland et al.00] Garland, Andrew; Lesh, Neal; Rich, Charles and Sidner, Candace. ‘Learning Task Models for Collagen’. Mitsubishi Electric Research Laboratory (MERL), USA. (2000)

[Garland et al.01] Garland, A.; Ryall, K. and Rich, C. ‘Learning hierarchical task models by defining and refining examples’. In First Int. Conf. on Knowledge Capture, Victoria, B.C., Canada. October, (2001).

[Garland02] Garland, Andrew and Lesh, Neal. ‘Learning Hierarchical Task Models By Demonstration’. Mitsubishi Electric Research Laboratory (MERL), USA – (January 2002)

[Maes94] Maes, Pattie. ‘Agents that reduce work and information overload’. In Communications of the ACM, Vol. 37, No 7 pp.30-40.(1994)

[Paterno et al.97] Paterno, F.; Mancini, C.; and Menicori S. ‘ConcurTaskTrees: A diagrammatic notation for specifying task models’. In S. Howard, J. Hammond, and G. Lindgaard, editors, Human-Computer Interaction INTERACT’97, pp. 362-369, Chapman and Hall. (1997)