Intelligent user task oriented systems*

Tiziana Catarci
Dipartimento di Informatica e Sistemistica "A. Ruberti"
Universita di Roma "La Sapienza"
Via Salaria, 113
00198 Rome, Italy
catarci@dis.uniroma1.it

Benjamin Habegger
Dipartimento di Informatica e Sistemistica "A. Ruberti"
Universita di Roma "La Sapienza"
Via Salaria, 113
00198 Rome, Italy
habegger@dis.uniroma1.it

Antonella Poggi
Dipartimento di Informatica e Sistemistica "A. Ruberti"
Universita di Roma "La Sapienza"
Via Salaria, 113
00198 Rome, Italy
poggi@dis.uniroma1.it

1. INTRODUCTION

Most existing human-computer interfaces are now based on the WIMP (Windows, Icons, Menus, Pointers) paradigm and on the desktop metaphor. Each application is targeted towards editing one specific type of "document". Current desktop oriented systems propose a mostly disconnected set of generic tools (word processor, e-mail reader, drawing tool, CD burning application, etc.). In such a setting, a user which is executing a specific task (e.g. making a CD from a set of files) will likely use more than one of these tools. The fact that they are running on one system without being connected leads to awkward situations such as the user reentering or copying data between applications. Furthermore, the same user is very likely to run multiple times the same task and often, she will do this in a similar manner. For example, a music fan will likely find himself burning more than one music CD. Having a system which is aware that some task is being run would allow for some anticipation and could relieve the user from much routine procedures.

To provide the user with complete power of her desktop computer, we propose the idea that she should be provided with a system having knowledge of her specific tasks. Such a task-oriented system should be designed to aid the user in managing *her* day-to-day tasks and helping her gain in efficiency. We consider that both tasks themselves and how they are to be lead are *specific* to each user. Building a system which is both adapted to the user and simple enough for being used by a non-expert user, is a challenge. At one extent, full adaptability is asking the user to specify herself her tasks. At the other extent, simplicity is having preprogrammed tasks not necessarily adapted to the user's needs. Therefore, having some form of adaptability, cannot go without at least some participation of the user in defining tasks. However, combing user interaction and inference methods (in particular machine learning) can help in making it easier for the user to provide task descriptions.

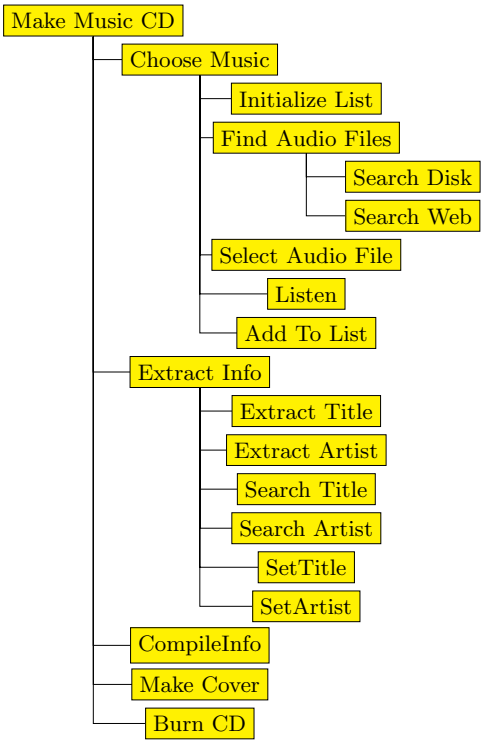

Figure 1: Example user-task decomposition

In this paper, we present our on-going work on user task-oriented systems. Note that this work follows up on the OntoPIM project [3], which has concentrated on the management of the user's data through a so-called *Personal Ontology*. The latter provides the user with a *virtual* description of her domain of interest, through which the user can access her personal *real* data. This offers a useful basis to define user tasks, in that the Personal Ontology can be used to specify how to map task input and/or output data (e.g. by posing an appropriate query over the Personal Ontology). The main contributions of this work are therefore the following. First, we define a task specification language, combining the simplicity of classical Hierarchical Task Analysis (HTA)[2], with the expressivity of a simple workflow language, in the style of YAWL [4]. As already mentioned, this language allows to possibly specify the task data flow in terms of queries over the Personal Ontology.

Clearly, in order for a specified task to be correctly executed, such a language should have a formal well defined semantics. Second, we show how, starting from a log that keeps a precise trace of the user's actions, it is possible to derive task specifications. In particular, for the task specification process to both be simple for the novice user and provide some power to the more expert user, we propose two different approaches to build a task specification by a combination of user interaction and techniques based on machine learning. The first approach allows the user to herself define a task decomposition. Then, given example runs of each subtask, the system is responsible for inferring missing parts of the overall task specification. On the other hand, in the second approach task-aware environment keeps track of user actions and identifies frequent patterns. Then, from these patterns new task specifications are proposed to the user, to gradually build up task hierarchies.

2. MOTIVATING EXAMPLE TASK

Let us consider a music fan who likes making personal music compilations based on a collection of audio files. This task, which we will call MakeMusicCD, usually follows some common steps such as looking for music audio files (e.g. in an appropriate directory of his hard drive), and selecting a subset of the music he has found. Also, whenever the user burns a CD he also likes to have the correct artist and title to be set (e.g. meta data associated to the audio files) and uses this information to make a cover for his CD. Using current systems our music fan finds himself using multiple completely decoupled tools such as a web search engine, a paint-type program, and a burning application. These applications share none of the information which is common to the MakeMusicCD tasks. Currently, unless a developer explicitly writes an application targeted to executing MakeMusicCD tasks, our music fan will likely not be able to automate much of his task.

The system we envisage has knowledge of the user's tasks and a hierarchical decompositions which depends on the user's view of each task. For example, the MakeMusicCD could be decomposed as shown in figure 1. Such decompositions can be obtained by following techniques such as hierarchical task analysis (HTA) [2]. The association of actions to leaf nodes of the hierarchical decomposition allows to at least partially, if not completely, automate the subtask. For example, a CD burning tool, might be associated to the "Burn CD" leaf subtask. Therefore, when the MakeMusicCD task is made active, the burning tool can at the very least be made ready. More automation can be obtained if the task execution system knows when and how the subtasks of a given task should be run. This requires having some form of plan defining how the subtasks of a node of the decomposition are to be executed. For example, a possible plan for the "Choose Music" task is given in figure 2. Further on, we describe two approaches which provide mechanisms for simplifying the construction of such plans.

By decomposing a task into a tree and associating actions to its leaves there is already a gain for the user in that he is no longer required to search for and individually start the required actions every time he performs his task. Running all of the leaf actions might not be the best solution but it is easy to imagine a system which proposes suggestions to

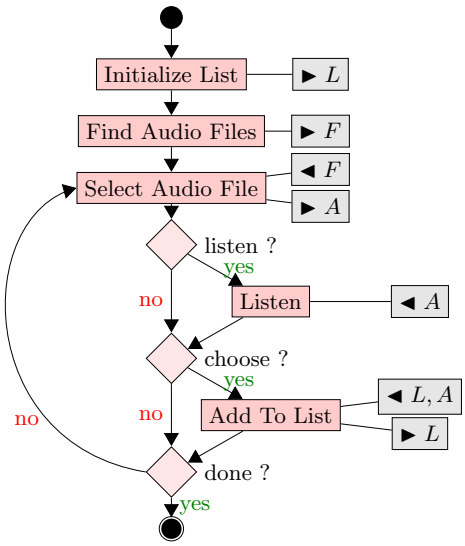

Figure 2: Plan for the "Choose Music" subtask

the upcoming actions (e.g. through a toolbar providing a list of possible actions). Having a plan associated to each node of the task decomposition allows the system to exactly know which subtask is to be run (and if the subtask is a leaf which tool should be started). This is yet a higher gain in efficiency for the user since in each task run the subtasks and associated tools can be started at the right time. Finally, if the system is also aware of how data is to be exchanged between subtasks, this will relieve the user from transferring the data himself (e.g. via copy/paste). If the system both knows when and with which data a subtask should be executed, then it could be simply be run without even requiring the user.

3. TASK SPECIFICATION LANGUAGE

In this section, we present the general ideas behind a task specification language for which we have defined formal syntax and semantics. Note that it is quite a challenge to define a task specification language which both provides the power necessary to define tasks having a real added-value for the user and, at the same time, which stays simple enough to allow task specification inference.

Our proposal is based on the idea of combining task decomposition and a *plan language* to describe for each task, the execution plan of its subtasks. Because of lack of space, we do not give here the details of our task specification language. Instead, we aim at giving an intuition of it. On one hand, a task is decomposed into a set of subtasks. This allows for a more comprehensible view of the task. On the other hand, we propose a plan language limited to sequence, alternatives and repetition. Our language is somewhat related to workflow languages used to model business processes such as YAWL [4]. The restrictions keep the language simple enough for inference to remain feasible and allows to have a quite straight forward semantics. Also, from the user's point of view, a useful benefit of such languages is that they have a natural graphical representation (as shown in figure 2) which is often more comprehensible for a non-expert user. Finally, we assume that data manipulated by a task may come from and possibly update the user Personal

Ontology. Also, our task specification language allows to link inputs and outputs not only with equality relationships (i.e. a subsequent action takes as its input a previous output as is), but also with queries over the user ontology. For example, given that action A has output a person instance p, a subsequent action B might be interested not in the person, but rather to p's director. This information being stored in the user ontology, the output query $DirectorOf(p, d)$ would be associated to the link between A and B.

One of the target applications is for the tasks to be partly *runnable* (ie. some subtasks can be executed without any user intervention). To be *runnable* only from the specification and input (and eventually ontology data), *all* the information required to make a task runnable (ie. both the flow of control and the flow of data) should be determined completely by the specification of a task. This is a challenge since most modeling languages (both workflow and dialog-oriented) are only abstractions. Furthermore, we likely do not want to make all possible constructs proposed available since this would require too much expertise from the user.

It should be emphasized that in this proposal while a task is runnable (i.e. the task management system keeps track of the current state of a running task), only *some* subtasks may be effectively automated. The termination of non-automated subtasks will be triggered by the user. For example, a subtask might be "call secretary to reserve a room" whose output would be a room and a date. This implies that the state of a running task will depend both on the user and the system. We will consider that a task *must* run as specified. Therefore, whenever a subtask terminates it is the system which determines the next step. However, a (sub)task may finish either because the user told the system that it was or because the system detected it. In both cases, this can be modeled as an event received by the task management system. (Note that, in this case, arriving in a final state is a *particular* termination event.)

We consider that a task will be defined as a tree where nodes are subtasks. The execution of a task consists in executing the root subtask. The execution of a leaf subtask is straight forward (running the associated action or waiting for a termination event provided by the user). The execution of a non-leaf subtask consists in running its child subtask following some plan associated to the subtask. We consider plans to be defined as a workflow using a set of nodes composed of a starting point, an ending point, the set of child subtasks and a set of decision nodes. The workflow will determine the flow of control among the different nodes (and in particular the subtasks). Of course the starting point has no incoming edge and the ending point has no outgoing edge. We limit subtask nodes or the starting point to only having one outgoing edge (ie. we enforce explicit decision nodes). Also each outgoing edge of a decision node has an associated condition function. Figure 2 tells us that the plan for "Choose Music" has a total of 9 nodes, among which the start node (represented by a black disk) and the end node (represented by a black disk with a circle around it), 5 subtasks (represented by rectangles) and two decision nodes (represented by diamonds). In this example, the decision nodes are interactive in that the choice taken depends on the user.

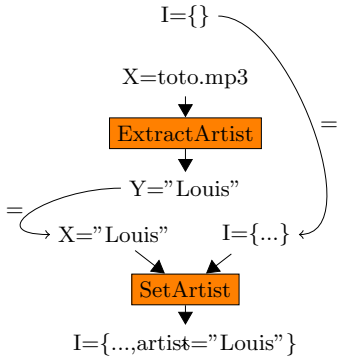

Figure 3: Example log of user actions

We also want to specify precisely how data is exchanged among the children of a running subtask. Currently, each subtask has a local memory and data is exchanged among the children only *via* this shared memory. This choice has been motivated by the fact, that definitions in which the data is transmitted directly between subtasks may yield problems namely due to the availability of data. One particular problem is that it is not evident how to (statically) check that, when the control is given to a subtask, it effectively has all the necessary data to run (in a single memory perspective, we only need to check that the used variables are defined). Being capable of proceeding to such checks when the user is defining his/her task allows to test for misconceptions. A second problem is to decide which data to be used in a particular run of a child subtask in the case a given input can be obtained from two distinct locations.

4. USER ACTION LOG

The system we propose is aware of user actions and moreover interacts directly with the user. This allows us to maintain a log of the user's different actions. This log provides us with the necessary information to help the user. A log gives the sequence of actions the user has done, their inputs and outputs and, finally, the links between the inputs and the output. The links define which particular output of previous tasks was an input coming from and how they are related through the ontology. This relationship can be obtained by appropriate user-interaction. Figure 3 gives an excerpt of an example task log which gives the trace of two actions "Extract Artist" and "Set Artist". The I variable whose value is an empty record at the top represents some information which was made available by a previous action (not necessarily immediately preceding those of the log). From the log we know that the "Extract Artist" action took place (just) before) the "Set Artist" action. Furthermore, we know that the Y output of "Extract Artist" action was used by the X input of the "Set Artist" action. The = sign on the arrow between these two variables indicates the relationship which held between the value from the output to the input (in this case equality). When proceeding to some action (say $WriteEmail$) using the output of a previous action (e.g. $FindCompany$) which generated a company instance, the user might have navigated through his/her personal ontology following a $DirectorOf$ relation and obtained a person instance. In this case, the $DirectorOf$ relationship would have replaced the = sign as the relationship between the output of $FindCompany$ and $WriteEmail$.

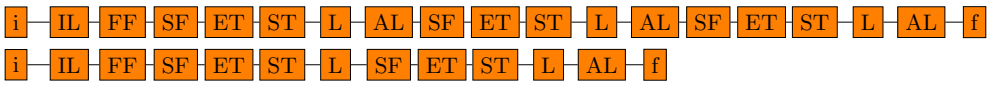

Figure 4: Example partial logs of two execution of a task

5. INFERRING TASK SPECIFICATIONS

One of the most challenging parts in managing the user's task is obtaining task specifications. Requiring that the user herself provides such specifications, which is the case in tools such as Apple's Automator [1], remains only feasible to users with some minimal expertise. However, by having a system which, since it is in charge of running the user's actions, may log these actions, inferring task specifications from these logs becomes possible. We currently envision two task inference mechanisms. The first allows the user to specify his own decomposition of a task and the role of the system is to infer the plan associated to each node of the decomposition. The second approach consists in inferring a task decomposition and the specification of a plan by an analysis of the user action log.

The first approach consists in having the user define a task decomposition by some technique similar to hierarchical task analysis (HTA). Once the decomposition is defined, the user provides executions of the tasks following his decomposition (simply by manually running them). The logs of the executions of a node is then used to infer its plan. For example, the user might have given the decomposition of figure 1. Multiple executions of the "Choose Music" node would have lead to different logs of this subtask. Since the list of subtasks is given by the decomposition, what remains to be determined is the connections between subtasks, the flow of data and the decision function associated to decision nodes (in the case they are non-interactive). The connections can be simply added when there exists a witness for them (i.e. a log which proves that a link does exist). The relationships which hold between inputs and outputs can be obtained by the log of the flow of data. Finally, the problem of determining a decision function can be transformed into a classification problem : given a set of variables which are available for the decision procedure, determine which state is next. Consider figure 4 which gives the control part of the log of two executions of a "Choose Music" subtask. Each subsequence of two subtasks of this log is a witness that the second subtask can be followed by the first. The names of the subtasks have been shortened to their capital letters. For example, by looking at the first log we can see that a "Listen" (L) subtask may be followed either by a "SelectAudioFile" (SF) or a "AddToList" (AL) subtask. Also, from the two logs we can also see that a task always starts with an "InitializeList" subtask.

The second approach allows to build both the user task decomposition (in a bottom up manner) and partial specifications for each task. Candidate specifications can be obtained by looking for simple (frequent) patterns and suggesting them as new tasks to the user. For example consider the "Extract Music Info" subtask in figure 1. When performing this task, the user repeatedly performs the same series of actions to each song. In addition, the log of the executions gives us the data connections between the actions. The combination of these allows us to recognize that the sequence comprises a potential task which can be proposed to the user. Furthermore a (partial) specification of tasks can be constructed. Once validated by the user, the newly recognized task becomes available as an action to the user and the inference engine. Therefore this new unit may appear in the logs and be recognized in patterns. In particular, we can then detect that it has been applied to each song and suggest a repetition task. Through these repeated steps of inference and user interaction we incrementally build complex task hierarchies.

6. CONCLUSION AND FUTURE WORK

In this paper we presented different aspects of a system allowing to manage a user's personal tasks. We proposed a task specification language based on a hierarchical decomposition of a task with a plan associated to each node. By having a system which has a direct access to the user's actions allows us to exploit the logs of these actions to either infer and propose new tasks specifications to the user and/or by inductive reasoning allow to help the user build the specification for previously given decompositions.

7. ADDITIONAL AUTHORS

Alan Dix
Computing Department
Lancaster University, Lancaster, UK
alan@hcibook.com
Yannis Ioannidis
Department of Informatics & Telecommunications
University of Athens, Athens, Hellas (Greece)
yannis@di.uoa.gr
Akrivi Katifori
Department of Informatics & Telecommunications
University of Athens, Athens, Hellas (Greece)
vivi@mm.di.uoa.gr
Giorgios Lepouras
Dept. of Computer Science and Technology
University of Peloponnese, Tripolis, Greece
gl@uop.gr

8. REFERENCES

[1] Apple automator. `http://developer.apple.com/documentation/AppleApplications/Conceptual/AutomatorConcepts/index.html`.

[2] Alan Dix, Janet Finlay, Gregory Abowd, and Russell Beale. *Human Computer Interaction*. Prentice Hall, 1998.

[3] V. Katifori, A. Poggi, M. Scannapieco, T. Catarci, and Y. Ioannidis. Ontopim: how to rely on a personal ontology for personal information management. In *Proc. of the 1st Workshop on The Semantic Desktop.*, 2005.

[4] W.M.P. van der Aalst and A.H.M. ter Hofstede. Yawl: yet another workflow language. *Information Systems*, (30):245–265, 2004.