Toward Tailored Information Presentation in Support of Collaborative Planning

Michael T. Cox

Department of Computer Science and Engineering
College of Engineering & Computer Science
Wright State University
Dayton, OH 45435-0001
mcox@cs.wright.edu

Abstract

In collaborative environments where multiple users interact, determining individual problem-solving contexts that include intent is difficult, even when individual user goals are known. Although intent may be explicit in a sub-context, often it is implicit and must be inferred. Moreover for both implicit and explicit intent or goals, actions to achieve such goals may interact negatively between users. This paper discusses two approaches to determining user and team intent and interactions. We look at a collaborative system that encourages user-specification of goals so that the system can track them, and we look at a keyhole plan recognition approach that infers user intent from individual user action. We enumerate some of the problems with both approaches.

Introduction

In general terms, we are interested in the problem of tailored information presentation. That is, we wish to understand how a system can present the right information to the a user at the right time and in the right format and modality when the user is in a collaborative environment (Brown & Cox, 1999). To begin to approach this problem we have examined a subset of this larger problem in the form of dynamically managing the user interface in support of user tasks. In particular we have built preliminary systems to solve simple screen clutter problems of multiple overlapping windows on a desktop environment. When the number of windows in the user-interface passes a given threshold, the task is simply to determine which window should be either moved, iconified, or killed. However, to perform such a task effectively, a system must know more than just screen dimensions and low-level window information. The system must be able to base a decision to change the user interface upon the user’s problem-solving context including the user’s intent.

Especially in collaborative environments where multiple users interact, determining individual problem-solving contexts is difficult, even when individual user goals are known. Although many user goals may be explicit in a sub-context, many are implicit and must be inferred. Moreover for both implicit and explicit goals, actions to achieve such goals may interact negatively between users. This paper discusses two approaches to determining user and team intent and interactions. We look at a collaborative system that encourages user-specification of goals so that the system can track them, and we look at a keyhole plan recognition approach that infers user intent from individual user action. We enumerate some of the problems with both approaches.

Tracking User Intent

Grosz, Hunsberger, and Kraus (1999) distinguish between two types of intentions. First an intention-that is defined as a desired state (goal or objective); that is it is a state of the world that an agent or user wishes to achieve. Secondly an intention-to is defined as a desired action; that is it is an action that the agent or user plans to perform. In most of this paper we use the word goal and intention-that interchangeably. When referring to intention-to, we will make such distinctions explicit.

GTrans (Cox, Kerkez, Srinivas, Edwin, Archer, 2000) is a mixed-initiative planning system that presents planning tasks to a user as goal manipulation problems rather than a problems of search (Cox, 2000). It operates in both the Unix and Windows environments. In the system a user performs direct drag and drop manipulation of objects on a graphical map. The user can set domain states and goals that are then sent to an automatic planner for results. In the current implementation, the underlying planner is a version of the PRODIGY state-space non-linear planner (Carbonell, et al., 1992;
Veloso, et al., 1995) called Prodigy/Agent (Cox, Edwin, Balasubramanian, & Elahi, 2001). This software communicates via sockets using a communication protocol specified in KQML. GTrans sends it a list of goals and an initial state defined by a set of objects and states; Prodigy/Agent sends back a stream of plans that transform the initial state into the goal state. The GTrans user can request different, shorter, and equivalent plans in the plan stream.

When the planner produces poor plans (from the user’s perspective) or when it cannot produce a plan at all, the user can manipulate goals in the system by performing various goal transformations (Cox & Veloso, 1998) upon them. For example in the logistics domain, the goal of having all packages in a city delivered in the morning may not be possible due to a delivery truck being repaired. The goal may be “reduced” to delivering most packages in the morning and the remainder in the afternoon (see Figure 1). In GTrans and similar applications, the user’s goals are explicit and constitute the focus of user behavior. The system does not need to infer intent; rather it need only track the user’s top-level goals\(^1\) to monitor intent.

A supporting system called P4P\(^2\) (Kerkez, Cox, & Srinivas, 2000) is a planner that plans for its own user-interface. As a human uses a planner such as GTrans, P4P “watches” the user interactions with the planning interface. After each user action, the current state of the interface represents a new initial state from which P4P will plan to reduce screen clutter. If no screen clutter exists, then no plan is needed. Otherwise P4P creates a sequence of interface actions such as (restore window1) (iconify window3) that can achieve the state of screen clarity. A simple planning domain called the micro-window domain has been created to model these behaviors.

We have identified three types of knowledge that can help a system reason about user interactions with user interfaces. First and most obvious is knowledge about window characteristics such as screen geometry and recency of window use (Kerkez & Cox, 2000). This knowledge represents a kind of window syntax. Second is knowledge about window content and function (Kerkez, Cox, & Srinivas, 2000). Such knowledge represents a kind of interface semantics. Third is knowledge about the user problem-solving context and represents conceptual knowledge about the reasons the user requires information transmitted through the interface.

The single-user problem-solving context is represented as the set of user planning goals (intent-that). In the simplest form, the screen clutter problem consists of matching the user goals to functions of objects in the user-interface. Given all other information (i.e., window characteristics and con-

---

\(^1\)A subgoal of a top-level goal is one that needs to be solved in order to solve the top-level one. For example to achieve the goal of having a package at a particular post office, the action to unload it from a truck in which it is contained will suffice. However if the loaded truck is not at that location of the post office, then a subgoal to have the truck at this location must be achieved before the top-level goal. GTrans does not automatically track subgoals.

\(^2\)Planning for Planners.
Multiuser Collaborative Planning

The GTrans system also operates in a collaboration mode whereby multiple users can each plan on a common map representation from different machines on the network. All users will view the same map and can each manipulate objects and goals independently. The results of individual interactions, however, are viewable to all users. For example a map may contain any number of regions in the logistics domain for which each user is responsible. As long as each user has independent top-level goals that pertain to their region only, then few planning problems exist. GTrans simply takes the intersection of all users’ states, objects, and goals, and it sends this to Prodigy/Agent. The plan that returns will be consistent with individual plans generated by each user.

However, interactions occur when the planned actions of one user interferes with the plans of another user. This may happen in the logistics domain when packages must be routed between regions and limited resources exist. Aircraft are shared by both users, so planning must be coordinated. In particular conflicting goals must be negotiated so that they “move closer” to each other to avoid conflict in a team collaboration.

In this kind of environment it is not sufficient to maintain the team problem-solving context (let alone the context of individual users) as the intersection of a top-level goals. Instead subgoal information is necessary. Because in a team environment interactions exist between team members, the interaction dependency should be represented in the problem-solving context to record which team member’s intended action (intention-to) will interfere with what other team member’s intention-that. As the P4P and GTrans implementation currently exist, no mechanism is present that can identify these team interactions.

A system called COMAS\(^3\) (Edwin & Cox, in press; 2001) allows multiple (artificial) planning agents to interact and create plans to achieve shared goals. Each agent is represented by the Prodigy/Agent automatic planning system. COMAS uses rationale-based planning monitors \(\text{(Veloso, Pollack, & Cox, 1998)}\) to identify negative and positive interactions between committed planning decisions as each pursues their respective planning goals. The system works by establishing a monitor for each state that is a dependency of some planning commitment. When another agent (say a team member) performs an action that results in negating the dependency state, the original agent must react to the change at the point the dependency was established in the search tree.

For example an agent may have two actions that can move a package between two points. One action is to fly a helicopter, and the other is to drive a truck. Because of speed the agent may decide to fly the only helicopter available. However if another agent decides to use the helicopter subsequently, and is able to “apply” the action to the current world state first (thus resulting in the unavailability of the helicopter), then the first agent must change the decision to drive a truck.

Note that by assuming the possibility of non-independent team member goals (i.e., interacting goals such as having two packages in two different locations with only one helicopter available), we are allowing a very dynamic team collaboration. This is not usually a feature of highly scripted team interactions where each member has clearly defined roles. In such cases where regularity is the norm, work-flow models of team behavior are more useful.

Inferring User Intent

When allowing for highly dynamic team behavior, forcing users to make all goals explicit and tracking goals with specialized software \(\text{(e.g., GTrans)}\) that users share are not always realistic strategies. In addition to planning with a collaborative system such as GTrans, users will also have supporting applications that comprise the larger planning context. For example external text editors and graphical displays may be involved to change plan sequences and visualize plan graphs. Geographically separated users may also use heterogeneous email applications to communicate. In such cases, the overall problem-solving context may be arbitrarily complex. To support such heterogeneous applications as email communication between team members, the problem-solving context should also include a record of existing information pipelines.

Plan recognition is an alternative approach to inferring user intent when a system has only observations of user actions and are not privy to internal operations of individual applications. The concept is to have a library of common actions in some domain and to match the current sequence of observation to the histories of known plans. Each plan then has associated goals which represent the intent. We have developed a novel method of plan recognition that combine intermediate states with action sequences to produce annotated plans. The plan recognition system then uses the current state as well as the latest set of actions to find previous plans matching isomorphic states. The problem is that such states may be quite large so the combinatorics are prohibitive. The solution we use is to represent states in an abstract representation that generalizes token to types, leav-

\(^3\)Coordination Of MultiAgent Systems.
ing a much smaller search space (Kerkez & Cox, 2001).

The plan library contains instances of plans an observed agent may pursue. Planning episodes in the plan library can be viewed as cases, and the recognition process can utilize these past cases to generate predictions of planner’s intentions (both intention-that and intention-to; i.e., both goals and actions). The case-based nature of the recognizer enables it to incrementally acquire planning instances, eliminating the need for hand-coded or automatic library construction methods. A pure case-based approach to plan recognition allows the recognizer to partially match cases (plans) from the library with its observations. The case-based plan recognizer can therefore recognize plans that are not exact matches to the plans from the library, but instead are similar to the description of a current situation, based on the observed planning steps.

At the current time, our plan recognition system does not observe user actions at the interface. Instead it operates in the blocksworld and logistics domains using a case library of plans generated by PRODIGY. Given a new set of observations (another instance of the planner’s behavior), the system retrieves old plans to make predictions of current behavior.

Observing User Actions

We are currently building two systems to trap user-interface events in arbitrary Unix and Windows desktop environments. CONDUIT works in a Unix desktop to both trap (i.e., record) and control (e.g., initiate window iconification) desktop events. UIA works in a Windows desktop to trap application events including both mouse and keyboard actions. We intend these systems to support the plan recognition system described above.

CONDUIT

The CONDUIT4 System constitutes a framework to both monitor these events and introduce triggered events in a standard deployment operating system desktop - in this case the GNOME Desktop running the Sawfish window manager (http://sawmill.sourceforge.net/). Sawfish is an X Windows Window manager that has the capability of being extended through a lisp dialect similar to elisp found in Emacs. By extending window manager events to provide notification to an external process, we can keep track of what a user is doing (like all other systems, there are limits).

Besides being able to track user events, CONDUIT provides a mechanism for the external process to send information back to the window manager. For instance, an external planner would gather information about the current status of windows on the user desktop and could rearrange the windows on a desktop to maximize information representation by providing commands for specific windows to Sawfish.

The base of the CONDUIT architecture is the interaction between the Window Manager and an external agent such as Prodigy/Agent or the plan recognizer. This piece of software is called CONDUIT/Monitor. CONDUIT/Monitor is the software that glues user interaction to the communication layer that provides information to the planner processes.

CONDUIT is comprised of a C program with a GTK+ GUI, and a series of functions that extend Sawfish’s functionality. The GUI for the CONDUIT/Monitor (as shown in Figure 2) was developed using a product called GLADE, which is a Graphical GTK+ GUI developer that can produce code in a variety of target languages including C, C++, Perl, and Allegro Common Lisp.

The code that is embedded in Sawfish is responsible for starting the external C program process and “printing” information we are interested in gathering to the C program that interprets the message and passes it to the external agent/planning processes. That external agent/planning process can then turn around and issue commands back through the CONDUIT process and into the window manager. This can be useful for starting external applications that can aid a user in a given task, or perform actions on windows, such as minimize the least recently used window, or other tasks.

The external process is an event loop driven system, using the Glib/GTK+ event loop driver, which accommodate file operation, pipes, and socket monitoring natively. - all the
programmer has to do is provide code to be run whenever that specific event is triggered.

**CONDUIT** focuses on X window manager events, and as such, does not reach into applications to monitor individual widgets or the state of the application. Some actions can be inferred by combinations of actions (e.g., inferring scrollbar movements by observing a mouse drag and no window movement) To monitor those application level events, one would require a version of the widget library (GTK+, Motif, and Qt) that has monitoring code embedded in it. This approach would still present problems due to how applications are linked to their respective widget libraries.

**CONDUIT** (as a whole) uses extensions of common event handlers to provide notification to the external CONDUIT process. This new behavior is added by adding the CONDUIT notification function to the list of functions called for a specific type of event in Sawfish.

**UIA**
The User Interface Agent (UIA) accomplishes the task of capturing the mouse and the keyboard events generated by a human user under the Windows operating system. The UIA operates at the operating system level, and it is not tied to a limited number of applications under the operating system, as it is commonly done. UIA is written in both Visual C++ and Java programming languages. The code that processes the low-level operating system events (e.g., user is creating a window) is written in C++, while the high-level interpretation of the events (e.g., user is opening a word processing application) is written in Java.

The low-level input events generated by the user are sent to the event queue associated with the operating system. The operating system then propagates the events into the application(s), where these events originated or were requested. The events can be intercepted and interpreted at the system event queue using the Windows API. A class of API functions, called hook functions, provide a mechanism of monitoring a variety of events known by the operating system. Windows supports many different types of hooks; each type provides access to a different aspect of its event-handling mechanism. For example, an application can use the WH_MOUSE and the WH_KEYBOARD hooks to monitor the events generated for mouse and keyboard events.

Like **CONDUIT**, UIA has a GUI that displays information for all applications (see Figure 3). The interface has a panel for mouse information, keyboard information, a list of all open windows, and information pertaining to the currently focused window. Unlike **CONDUIT**, UIA does not currently have the ability to send control information back to the desktop environment.

The low-level events must then be interpreted in order for the UIA to create the world state information. Which hook functions need to be installed depends on a kind of user interface manipulation the reasoning system utilizing the intercepted events will need. The primary beneficiaries of UIA’s capabilities are incremental state-space plan recognition systems, as described in (Kerkez & Cox, 2001). Therefore, UIA must consider windows movement, size change, iconification, and similar events as the events which change the state of the user interface when the position of the windows is the most important user interface feature. After a state-changing event is observed, UIA dispatches the newly achieved user interface world state, along with the event itself to the plan recognizer. Since low-level events may include the information that is redundant for the task in hand, UIA’s task is to filter the information received and create the world states when appropriate events arrive.

The particular representation for interface states in the screen clutter problem (or the tailored-information problem in general) is an open question. In Kerkez and Cox (2000), we represented the spatial position of windows as being fully in one of four screen quadrants. This crude level of granularity is obviously insufficient for all but proofs of concept. However to represent locations in relative pixel coordinates given particular screen resolutions, requires a state space that is unacceptably large. Many such questions exist for future research before such systems can be fully integrated, especially when trying to combine low-level inference through observation and high-level inference that depends upon special constraints that either the user’s task or special software provides.

**Conclusion**
The purpose of this paper is not to present a finished research effort nor to report results that directly pertain to team intent inference. Indeed the representation of team intent is not simply the concatenation of individual team member intentions (this paper has not even attempted to address this
Rather the paper's purpose is an attempt to begin to associate a set of research tools and representations that can examine the issues surrounding an incredibly complex problem. The problem is to support multiagent teams (whether human, artificial, or mixed-agent teams) that together must collaborate to achieve some set of goals by providing the team members with the proper kind of information to enable them to solve their part of an overall problem without interfering with each other. Although we believe that determining the goals and intent of the team is a crucial piece of knowledge that must be determined if this support is effective, at this point, we are simply beginning the process of forming some of the important questions that surround the issue.

Acknowledgments

This paper is supported by the Dayton Area Graduate Studies Institute (DAGSI) under grant #HE-WSU-99-09, by the Air Force Office of Scientific Research (AFOSR) under grant number F49620-99-1-0244, and by a grant from the Information Technology Research Institute (ITRI). The author gratefully acknowledges Boris Kerkez and Jeremy Gwinnup for their work on UIA and CONDUIT respectively and their descriptions of the operation of these systems as described in this document.

References


