Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computers in Human Behavior 22 (2006) 709–732
Computers in Human Behavior
www.elsevier.com/locate/comphumbeh

A framework for specifying and monitoring user tasks

Brian P. Bailey *, Piotr D. Adamczyk, Tony Y. Chang, Neil A. Chilson

Department of Computer Science, University of Illinois, 201 N. Goodwin Avenue, Urbana, IL 61801, United States

Available online 8 February 2006

Abstract

Interrupting users engaged in tasks typically has negative effects on their task completion time, error rate, and affective state. Empirical research has shown that these negative effects can be mitigated by deferring interruptions until more opportune moments in a user's task sequence. However, existing systems that reason about when to interrupt do not have access to models of user tasks that would allow for such finer-grained temporal reasoning. To enable this reasoning, we have developed an integrated framework for specifying and monitoring user tasks. For task specification, our framework provides a language that supports expressive specification of tasks using a concise notation. For task monitoring, our framework provides an event database and handler that manages events from any instrumented application and a task monitor that observes a user's progress through specified tasks. We describe the design and implementation of our framework, showing how it can be used to specify and monitor practical, representative user tasks. We also report results from two user studies measuring the effectiveness of our existing implementation. The use of our framework will enable attention aware systems to consider a user's position in a task when reasoning about when to interrupt.

Keywords: Attention; Interruption; Task models; Task monitoring

* Corresponding author. Tel.: +1 217 333 6106; fax: +1 217 244 6869.
E-mail address: bpbailey@uiuc.edu (B.P. Bailey).

0747-5632/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.chb.2005.12.011

1. Introduction

When applications interrupt users at less opportune moments in their task sequence, disruptions to task performance (Bailey, Konstan, & Carlis, 2001; Czerwinski, Cutrell, & Horvitz, 2000b; Monk, Boehm-Davis, & Trafton, 2002), error rate (McFarlane & Latorella, 2002), and affective state (Adamczyk & Bailey, 2004) are much more severe than if the interruption had occurred at a more opportune moment. Prior work has both argued (Miyata & Norman, 1986) and empirically demonstrated (Adamczyk & Bailey, 2004; Iqbal, Adamczyk, Zheng, & Bailey, 2005) that subtask boundaries during task execution represent more opportune moments for interruption than non-boundary moments. One explanation, among others, is that a user's mental workload temporarily decreases at boundary moments (Iqbal et al., 2005; Iqbal, Zheng, & Bailey, 2004), leaving more mental resources for the interrupting task and for later resuming the previously suspended task.

These and other empirical findings have created rapidly growing interest in developing attention aware systems that can computationally balance a user's need for minimal disruption with their desire for information. An empirically supported approach is to defer the delivery of information, such as email notifications, system alerts, and instant messages, until a user reaches an opportune moment in a task sequence (Bailey & Konstan, 2005; Horvitz, Jacobs, & Hovel, 1999). In office settings or other work environments where peripheral information is often desired, but not generally safety critical, this approach could allow users to exchange awareness of information for mitigation of disruption.

We have developed a task specification and monitoring framework that facilitates the creation of such attention aware systems. Our framework consists of four components; a task description language that supports expressive specification of tasks using a concise notation, a graphical tool that enables rapid assembly of task specifications, an event database and handler that manages user events from instrumented applications, and a task monitor that follows a user's progress through specified tasks, notifying user-level services when task-related events occur.

Existing systems that reason about when to interrupt users rely on external and non-task specific cues (Horvitz, 1999; Hudson et al., 2003). By supporting models of tasks informed by and consistent with prior empirical work (Adamczyk & Bailey, 2004; Bailey & Konstan, 2005; Zacks, Tversky, & Iyer, 2001), our framework enables systems to draw upon this knowledge when making decisions about when to interrupt. While there has been work on task description languages for generating interfaces (Szekely, Luo, & Neches, 1993), predicting usability (Card, Moran, & Newell, 1983; John, 1995; Kieras, Wood, Abotel, & Hornof, 1995), guiding cognitive models (Ritter, Baxter, Jones, & Young, 2000), and research on task monitoring by cooperative agents (Franklin, Budzik, & Hammond, 2002; Rich & Sidner, 1998), our work provides an integrated framework for both specifying and monitoring user tasks. Rather than infer task models from user events (Maulsby, 1997), our framework includes a suite of effective end-user tools for rapidly creating task specifications and then monitoring those tasks during execution.

An important contribution of our framework is that it provides an open architecture, enabling tasks involving any application with appropriate instrumentation to be monitored and any user-level service to be notified when task-related events occur. Our frame-

work thus enables systems to have access to accurate information about a user's current position in a task sequence, important for intelligent tutoring systems (Cheikes et al., 1998), software agents (Lieberman, 1997; Maes, 1994), and attention aware systems that manage interruption (Horvitz et al., 1999).

2. Related work

We review empirical evidence showing that interruptions have a negative impact on users and their tasks and discuss how attention aware systems can leverage task models to mitigate those effects. Then, we discuss how existing task description languages, scripting languages and frameworks, and task monitors are not sufficient to operationalize these empirical findings, and explain how our framework builds upon this prior work to move closer to this goal.

2.1. Interruption and task models

Many experiments have shown that interrupting users engaged in tasks can have a significant, negative impact on task completion time (Cutrell, Czerwinski, & Horvitz, 2001; Czerwinski, Cutrell, & Horvitz, 2000a, 2000b; McFarlane, 1999; Monk et al., 2002), error rate (Latorella, 1998), decision-making (Speier, Valacich, & Vessey, 1999), and affective state (Bailey & Konstan, 2005; Zijlstra, Roe, Leonora, & Krediet, 1999). To mitigate effects of interruption, Miyata and Norman (1986) have speculated that task (and subtask) boundaries represent more opportune (or less disruptive) moments for interruption since users have reduced mental workload at those moments. They argue that when a user completes a task, the executive system releases the mental resources allocated for performing the task, momentarily reducing workload before the cycle of allocation and deallocation occurs again for the next task.

Experiments have empirically supported this speculation. Bailey and Konstan (2005) and Iqbal et al. (2005) showed that delivering peripheral tasks at particular boundaries during task execution causes considerably less disruptive impact than at other moments in the task. Since a small deferral resulted in a large mitigation of disruption, these results show that temporal manipulation of information offers an effective and practical computational strategy for mitigating effects of interruption. Our work seeks to enable such computational strategies by developing a language for specifying task models and marking moments selected for interruption and by developing a task monitor that allows higher-level services to defer delivery until those selected moments are reached.

For human–computer interfaces, a task model represents the hierarchical and sequential structure of a task (Card et al., 1983). Task models link how a person cognitively structures a task (what to do) with the actions afforded by a particular interface (how to do it). Models can be constructed by applying task modeling techniques such as GOMS (John & Kieras, 1996) or event perception theory (Zacks et al., 2001). For example, a typical scenario of use for GOMS is to develop initial models for a set of interface tasks, refine the models based on observing users performing the tasks, and then validate the refined models by observing another set of users performing the same tasks. However, the formality applied depends on the desired accuracy of the models. When creating task models, some form of a description language is almost always needed to express and represent the models.

2.2. Task description languages

A task description language provides a formal syntax and semantics for creating task models. The constructed models can then be used to specify and communicate interface designs, generate interfaces, predict the usability of interfaces, or enable systems to monitor user activities.

For specifying interface designs, description languages include task grammars (Shneiderman, 1982), modeling notations (Carr, 1994; Hartson, Siochi, & Hix, 1990; Siochi & Hartson, 1989; Tauber, 1990), algebraic specifications (Guttag & Horning, 1980), and transition diagrams (Harel, 1987). If expressive and detailed enough, the models can even be used to generate an executable form of the interface (Szekely et al., 1993). However, the models constructed with these languages would not generally allow a system to monitor the execution of the tasks.

Research in cognitive modeling has produced several task description languages, e.g., those used in (Byrne & Anderson, 1998; Byrne, Wood, Sukaviriya, Foley, & Kieras, 1994; John, Vera, Matessa, Freed, & Remington, 2002; Kieras & Meyer, 1997; St. Amant & Riedl, 2001). Once developed, the models can be typically passed to a cognitive simulator to predict usability (Ritter et al., 2000). For example, GLEAN (Kieras et al., 1995) offers an English-like syntax for describing the hierarchical, sequential and unordered parts of a task. An author uses the language to describe fine detail of an interaction such as 'move hand to mouse', 'move cursor to location', and 'click button,' which is necessary for the simulator to make predictions. These types of task description languages have been used successfully to build models of complex interface tasks and have led to design improvements (Gong & Kieras, 1994). While useful for simulating performance, these languages are more complex and require more specification detail than what interruption management would probably require, as indicated in (Bailey & Konstan, 2005).

Task description languages have also been created to allow software agents to monitor user activities. For example, to apply discourse theory to human–agent interaction, Rich and Sidner (1998) developed a task description language that allowed agent behavior to be linked to specific actions in the interface during design. As part of the Intelligent Classroom, Franklin et al. (2002) developed a task description language that allowed an agent to monitor an instructor's tasks and cooperate by managing the media capture devices. Each language has elements that would be useful for attention aware systems, but the languages themselves are inextricably tied to the particular system, which severely limits the ability of others to build upon their implementation.

As part of a project on embedded training, Cheikes et al. (1998) developed a task description language that allowed context-specific instructions to be integrated into task models at multiple levels of detail. While the description language used constructs similar to our own language (e.g., InOrder and AnyOrder tags) for expressing patterns of interface events, the monitoring system did not support multi-tasking behavior, which is common in practice.

While task models are usually constructed through manual use of description languages, there has been work to automatically infer the models. For example, ActionStreams (Maulsby, 1997) is a system that attempts to inductively learn the hierarchical, sequential, and variable parts of a task model from the user event stream. This is done by learning a grammar that expresses the sequences of incoming events. Maulsby (1997) acknowledges that learning a grammar is a complex problem and, in some cases, no algo-

rithms have yet been discovered that would enable the system to function as desired, e.g., learning grammars for arbitrary interleaving of events. Though learning task models is attractive, it is beyond the current state of the art for general use.

2.3. Scripting languages and event frameworks

Scripting languages and event (or message passing) frameworks have been developed to enable system-wide communication among applications and to support advanced functionality within individual applications. System-wide frameworks such as AppleEvents (AppleScript) typically provide a centralized communication manager that enables applications to publish and subscribe to registered events and exchange data. When an application publishes an event, the communication manager notifies applications that previously subscribed to the event by invoking a callback routine. The scripting language, e.g., AppleScript, is typically provided as part of the framework and can be used to program the desired response behavior.

Within applications, scripting languages enable sequences of interface commands to be recorded as macros, which are executable descriptions of a task. For example, Adobe Photoshop enables a graphic designer to visually record a sequence of image editing operations, edit the sequence, and then execute it on batches of images.

While existing scripting languages and frameworks support the exchange of *individual* events, they do not support explicit structures of task models or notifications of task-related events, e.g., that a user just crossed a particular subtask boundary. However, system-wide frameworks such as AppleScript could be used to facilitate implementation of a framework similar to our own.

2.4. Task monitoring

Many systems, e.g., (Cheikes et al., 1998; Franklin et al., 2002; Maglio, Barrett, Campbell, & Selker, 2000; Maulsby, 1997; Rich & Sidner, 1998) monitor the user event stream and compare events to a task model in order to provide context-sensitive instruction or feedback. While our system provides similar function, it also attempts to learn a flexible model of task execution and record that model in a user profile. In other words, the specified task model describes all the possible sequences of events and the model of task execution describes how often a user has historically followed each of those sequences when executing the tasks.

Bayesian networks have been applied to infer a probability distribution over user tasks (Albrecht, Zukerman, & Nicholson, 1997; Horvitz & Apacible, 2003). The networks typically use specific events or properties of events as evidence variables. This works well for identifying a task in the midst of sparse or noisy data. However, Bayesian networks by themselves cannot easily monitor multiple instances of the same task (e.g. preparing two separate email messages) or multiple active tasks (e.g. interrupting the editing of a document to send an instant message and then resuming), both of which are common in multi-tasking environments. Also, building or adapting the computational machinery for a Bayesian network would be overly difficult for most.

A challenge in task recognition is how to handle situations where multiple tasks match the same initial sequence of events. In this case, our task monitor maintains a candidate set of possible tasks and refines the set as more events are generated. While our approach

provides a working solution, more sophisticated, probabilistic approaches such as Dempster–Shafer theory (Carberry, 2001) could be used in the future. User preferences for execution sequences and more domain specific information could also help resolve ambiguity in task recognition.

To summarize, our task description language extends prior work in that it leverages the syntactic structure of XML to more easily support hierarchical decomposition, it draws upon the use of regular expressions to describe patterns of events in a concise notation, and it results in task models that are reasonably easy to read and understand. Our task monitor extends prior work in that it can monitor multiple ongoing tasks and multiple instances of the same task, seeks to learn a model of task execution, and uses a client/server architecture to support multiple applications.

3. Framework design goals

Several design goals were defined to guide the development of our task specification language and monitoring system. The term *author* refers to the person writing a task specification, which could be an interface designer, developer, IT support staff, end user, or other stakeholder. The goals of the system are to:

- *Enable low-investment creation of task specifications*. The benefit that comes from specifying tasks should ostensibly outweigh the investment required to specify those tasks. While we have shown the potential benefit of task monitoring for attention aware systems (Adamczyk & Bailey, 2004), realizing a net benefit requires a task specification language that is reasonably easy to use and learn and that is accompanied by effective interface tools.
- *Enable tasks to be specified at multiple levels of detail*. For example, a compose email task could be decomposed into open window, compose and send mail subtasks. Compose could then be further decomposed into select recipients, enter subject, and enter body subtasks, and so forth. For attention aware systems, finer-grained task decomposition would enable finer temporal reasoning about when to interrupt (Adamczyk & Bailey, 2004), but also requires more effort on part of the author. Striking the appropriate balance between level of detail and specification effort should be left to the author's discretion, not imposed by the system.
- *Support expressive descriptions of tasks*. An effective language should enable an author to express variations of task execution in a concise notation. Although there may be many different execution sequences to accomplish a task, an author should not have to explicitly describe all those variations; rather the language should accommodate multiple interpretations. This is analogous to how regular expressions provide a notation that enables a single specification to describe several matching patterns of strings.
- *Enable specification of tasks that involve multiple applications*. Interactive tasks often involve multiple applications. An example is that a user receives an email with an attached document, opens the document, edits it, and emails it back to the sender. If performed often, an author may want to specify this sequence as a single task because it provides a more accurate representation of the interaction sequence.
- *Accurately monitor specified tasks in the midst of unspecified activities*. Due to the enormous number and diversity of tasks possible in a typical computing environment, a task monitoring system cannot expect that every task that a user performs would

have an associated specification. However, research shows that users often spend about 81% of their time performing core tasks in a few applications (Czerwinski, Horvitz, & Wilhite, 2004). Thus, even if a system is able to monitor only a small part of the overall task space, it is still possible for it to recognize tasks that a user performs most of the time.

- *Support forecasting of a user's task execution*. By building a model of how a user performs and transitions among specified tasks, a system could forecast the user's task execution. The temporal granularity of the forecasting would be commensurate with the level of detail in the specifications. For example, for a compose email task specified at a coarse level, a system could forecast that a user will spend 5 min composing an email, or if specified at a finer level, that the user will spend 1 min selecting recipients, 30 s writing the subject, and 3.5 min composing the body. Forecasting would be useful to enable an attention aware system to better reason about when to interrupt a user engaged in a task, e.g., by deferring the delivery of information until the user reaches a boundary in their task sequence (Miyata & Norman, 1986).

While our current implementation does not yet fully achieve all of these goals, we felt it was vital to define them up front and use them to guide implementation and other design decisions.

4. Framework architecture

As shown in Fig. 1, our framework consists of four components; (i) a task description language that enables an author to express tasks at multiple levels of detail, (ii) an event database and handler that manage user events, (iii) a graphical tool called PETDL Maker that can be used to quickly assemble task specifications, and (iv) a task monitor that follows a user's progress through specified tasks, recording transition frequencies and the time spent at each step.

In our framework, the term *event* refers to an application-level event, which is a system-level event that has been delivered to and interpreted by an interface control. For example, a system-level event is 'mouse click' while an application-level event is 'select file menu.' Our framework assumes and only ever receives application-level events that were generated by user interaction. Our framework further assumes that applications have been instrumented to send these events. To demonstrate the feasibility of and test our framework, we instrumented MS Outlook, MS Word, and Firefox using their built-in scripting tools to generate events for common tasks. In the future, we expect that applications would have such instrumentation already available.

The framework uses a client/server architecture where the event handler and task monitor execute in a server process on the same or separate machine. Executing the server on a separate machine would enable the task monitor to monitor the tasks of multiple users simultaneously and across heterogeneous systems.

4.1. Task description language

Pattern-based Event and Task Description Language (PETDL) is an XML-based language for describing user tasks that draws upon GOMS (Card et al., 1983), regular expressions, and schema descriptions. Table 1 shows the tags available in the language

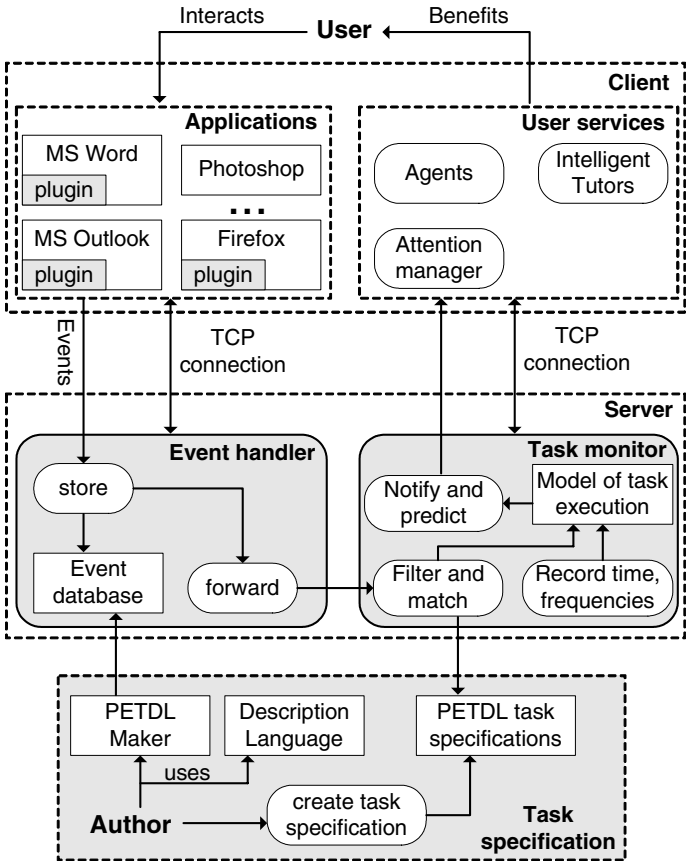

Fig. 1. Our framework consists of a task description language, an event database and handler, a graphical tool called PETDL Maker for assembling task specifications, and a task monitor. The framework uses a client/server architecture to communicate with applications and services.

and Fig. 2 shows how a calendaring task in Microsoft Outlook could be specified using the tags. Any number of task specifications can be contained in one PETDL document. PETDL includes tags to describe events, hierarchy, references, and pattern matching:

- *Events*. An author uses the ⟨`event`⟩ tag to name an application-level event in a specification. Although not shown in Fig. 2 for brevity, event tags include attributes for specifying the name of an application. By including the application name with an event, specifications can include events from multiple applications. The name of the event must exactly match the name of the event forwarded. Because an application registers an event dictionary with our event database, an author is able to view all events available for supported applications when writing specifications.
- *Hierarchy*. Drawing upon task modeling techniques such as GOMS (Card et al., 1983), PETDL enables an author to hierarchically decompose a task into subtasks (goals) and patterns of events (operators). PETDL provides a single tag ⟨`task`⟩ that can be recursively nested to more easily express hierarchy. The use of nesting allows –

Table 1
Tags available in our task description language

PETDL Tag	Description
Task	Expresses hierarchical task structure and enables reuse
inOrder	Children tags must occur in specified order
anyOrder	Children tags can occur in any order, but all must occur
Optional	Zero or one of specified children may occur
oneOrMore	One or more of specified children may occur
zeroOrMore	Zero or more of specified children may occur
repeatExactly	Children must occur an exact number of times
Choice	Exactly one of the children may occur
Event	An application-level event

```
<task name="Manage Schedule">
 <task name="Schedule Appointment From Email">
  <inOrder>
   <event name="OpenMailItem"/>
   <optional>
    <event name="SwitchFocus"/>
   </optional>
   <anyOrder>
    <task name="AddAppointment">
     <inOrder>
      <event name="OpenApptItem"/>
      <oneOrMore>
       <event name="ChangeApptItemProp"/>
      </oneOrMore>
      <event name="WriteApptItem"/>
      <event name="CloseApptItem"/>
     </inOrder>
    </task>
    <event name="CloseMailItem"/>
   </anyOrder>
  </inOrder>
 </task>
</task>
```

Fig. 2. Sample specification for a calendaring task using PETDL.

rather than forces – tasks to be specified at multiple levels of detail. In a ⟨task⟩ tag, any number of control tags can be nested to express patterns of matching user events. The names of the control tags were designed to reflect the event patterns they express. Also, the ⟨task⟩ tag supports a reference attribute that allows it to be named and then reused elsewhere in the same or other specification. The hierarchical structure of the model implicitly defines boundary points, with each end tag for a task defining a boundary. The nesting level of the boundary is used to infer how opportune that moment would be for interruption. Alternatively, an author can set the opportune attribute that takes an integer parameter, with lower numbers being more opportune for interruption.

- *References*. A positive consequence of using nesting to express hierarchy is the ability to rapidly create new specifications by composing reusable parts of existing ones. This reduces duplication of common subtasks, making them easier to maintain, and enables specifications to be shared among authors. PETDL allows a reference to be made to existing tasks/subtasks through the use of a `ref = true` attribute in a ⟨`task`⟩ tag. For example, in Fig. 3, the `AddAppointment` subtask is referenced and reused later in the task specification, and could also be used in other specifications.
- *Pattern matching*. Listed in Table 1, PETDL provides seven control tags for specifying rich patterns of user events. Control tags are built to resemble the control syntax used in regular expressions, e.g., the use of ⟨`zeroOrMore`⟩ in PETDL is equivalent to the use of an asterisk (*) in regular expressions. Similar to describing matching patterns of strings, an author uses control tags in a task specification to concisely express matching sequences of user events. Beyond regular expressions, however, our language also includes an ⟨`anyOrder`⟩ tag, as this tag allows many variations of execution sequences to be immediately expressed. For example, the use of this tag in Fig. 2, states that a user may either, add the appointment then close the email, or close the email and then add the appointment. Just as with regular expressions, some patterns of events may be described with alternative combinations of control tags. We leave the decision of how to best express patterns of events up to the specification author.

4.2. Event database and handler

The event database maintains a persistent store of event dictionaries for applications and records live streams of events generated by a user interacting with instrumented appli-

```
<task name="Manage Schedule">
 <task name="AddAppointment">
  <inOrder>
   ...
  </inOrder>
 </task>
 <task name="Schedule Appointment From Email">
  <inOrder>
   ...
   <anyOrder>
    <task name="AddAppointment" ref="true"/>
    <event name="CloseMailItem"/>
   </anyOrder>
  </inOrder>
 </task>
</task>
```

Fig. 3. A specification using a task reference. References allow tasks and subtasks to be reused in the same or other specifications.

cations connected to our server. When an application starts, it can connect to the event database to register or later modify its event dictionary. An event dictionary gives the name and description of all events that can be generated by an application. An author can inspect the event dictionary directly (as it is plain text) or can use our PETDL graphical tool which supports features such as filters.

After registering or modifying its event dictionary, an application connects to the event handler executing in our server. The event handler manages the streams of events generated by users interacting with the applications. For each event, the name of the event, the application that generated it, and the time that it was generated are sent to the event handler. The handler forwards this event structure on to the task monitor and then records it into the database. Because events are recorded, an author can use our graphical tool to monitor live streams of events and use these events to author task specifications in a way similar to macro authoring. A stream is removed from the database once the corresponding application is exited.

4.3. PETDL Maker

PETDL Maker is a graphical tool written in Visual Basic that enables quick assembly of task specifications. As shown in Fig. 4, the tool enables an author to view elements from

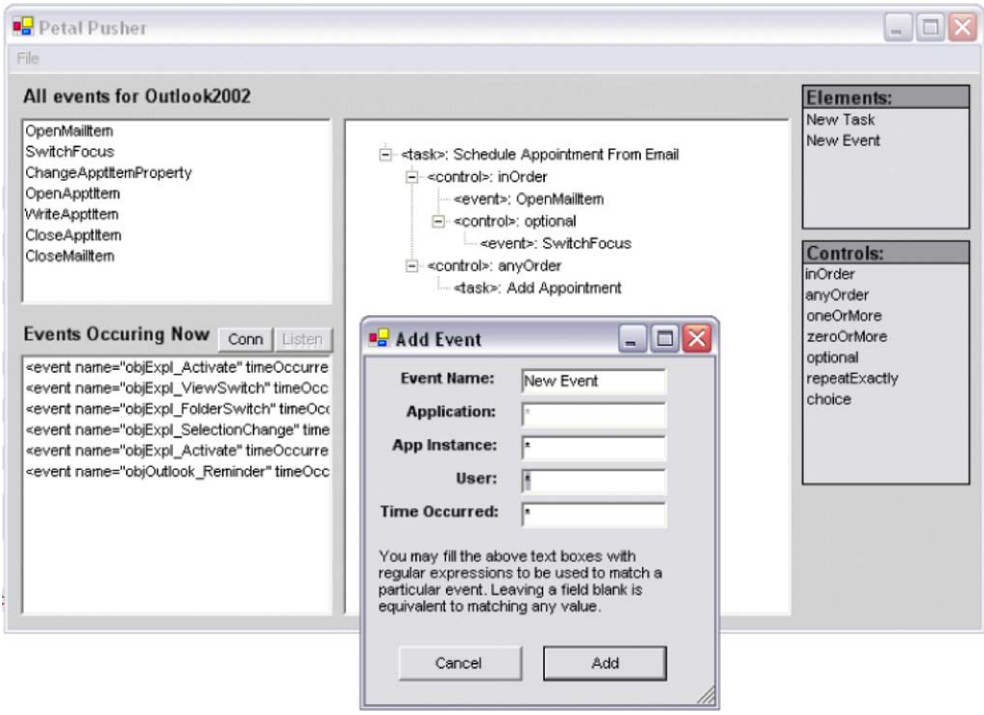

Fig. 4. The PETDL graphical tool used to quickly assemble task specifications. Events from the event dictionary are listed in the upper left while events from the incoming event stream are listed in the lower left. Control tags are listed at the right. An author may drag events and control tags from any of these sources and drop them into the document region (middle). The dialog shown in the center can be used to manually add events to the specification.

event dictionaries, the live stream of events being generated by user interaction, and the control tags. Because the list of incoming events can be large, the tool supports filtering based on application name and time. To create a specification, an author drags elements from any of these sources and drops them into the document region. The elements are inserted into an editable tree-structure that reflects and enforces the hierarchical nature of the language (see Fig. 4). Our experience with the tool shows that the ability to monitor events while interacting with an application is particularly useful for creating specifications. For example, to create a specification for a calendaring task, an author would run the PETDL graphical tool, perform the calendaring task as usual, view the live stream of events being generated by the interaction, and then select the desired events and appropriate control tags to assemble task specifications. We believe that this style, akin to macro authoring, will facilitate quick and accurate creation of specifications.

4.4. Task monitor

The task monitor follows a user's progress through task specifications and notifies user-level services of task-related events, e.g., starting or finishing a task. The task monitor receives events from the event handler and matches them to the available specifications, loaded at startup. The algorithm for matching incoming events to specifications is shown in Fig. 5.

```
HandleEvent(event)

matched ← False

« check against existing position placeholders »
for (p in placeholders)
   if p is older than 1 hour
      delete p from placeholders
   else if p.matches(event)
      p ← p.handleEvent(event)
      matched ← True

      « check to see if the task is completed »
      if p is NULL
         delete p from placeholders

« see if it's a new task »
if not matched
   for (t in tasks)
      if t.matches(event)
         p ← t.handleEvent(event)

         « add the new position placeholders to the existing list »
         placeholders.add(p)
         matched ← True

« if matched is False, then the event has been ignored »
```

Fig. 5. Algorithm for managing the position placeholders in the task monitor.

The task monitor maintains a list of position placeholders, which is initially empty. A *position placeholder* points to the next matching event or control tag in a specification and there is one placeholder for each instance of a task that the user is currently in. This allows the user to be in the midst of multiple tasks or the same task multiple times, an important advantage over many existing approaches, such as those discussed in (Cheikes et al., 1998; Horvitz, Breese, Heckerman, Hovel, & Rommelse, 1998).

When an event arrives, each placeholder (or active task) attempts to match the event to its next allowable events. For example, if a placeholder points to an `AnyOrder` tag, it would match the event against its contained events and recursively against the next allowable events of its contained control tags. If a placeholder matches an event, it then *handles* the event. Handling an event involves marking the matched event as having occurred in that instance of the specification and determining if it was the last event of the tag. If it was the last event, a placeholder to the next control tag is returned, otherwise the same placeholder is returned.

If the incoming event does not match any existing placeholders, the event is compared to events that start new tasks. If it matches, a new placeholder is created and added to the list. Otherwise, since the event does not move an existing task forward or start a new task, it is discarded. This process repeats itself for each incoming event. If a placeholder has not moved for a specified duration, then it is removed from the list and the specification must be matched from the beginning. This situation may occur if a task was interrupted and never resumed.

To demonstrate the algorithm, suppose the event stream in Fig. 6 is matched against the Manage Schedule task in Fig. 2:

1. This first event, `AppActivate`, is ignored by the task monitor because it does not match the first event in the task description, `OpenMailItem`. The task monitor continues to ignore events until it sees `OpenMailItem`.
2. The second event, `OpenMailItem`, matches the first event of the `Schedule Appointment From Email` task. Since this is the first event in the task, the task monitor creates a position placeholder to track progress through the task. The next matching event can be `SwitchFocus`, which is optional, or `OpenApptItem` or `CloseMailItem`, which can occur in any order but both must occur.

```
1. <event name="AppActivate"
        timeOccurred="2004.07.04 18:36:09" />
2. <event name="OpenMailItem"
        timeOccurred="2004.07.04 18:36:30" />
3. <event name="ReadMailItem"
        timeOccurred="2004.07.04 18:36:30" />
4. <event name="OpenApptItem"
        timeOccurred="2004.07.04 18:37:53" />
5. <event name="ChangeApptItemProp"
        timeOccurred="2004.07.04 18:38:59" />
6. <event name="ChangeApptItemProp"
        timeOccurred="2004.07.04 18:38:59" />
```

Fig. 6. Example event stream with events numbered for reference.

3. The third event, `ReadMailItem`, does not match the existing position in the task nor does it start a new task. Since it does not match, it is ignored by the task monitor.
4. The fourth event, `OpenApptItem`, matches the position placeholder and now the position placeholder is pointing to `ChangeApptItemProp`.
5. The fifth event, `ChangeApptItemProp`, matches the position placeholder. It is updated and now expects another `ChangeApptItemProp` or `WriteApptItem`.
6. The last event, `ChangeApptItemProp`, also matches the position placeholder because of the ⟨`oneOrMore`⟩ control tag. The position placeholder will still match more `ChangeApptItemProp` events or a `Write-ApptItem` event.

Because the algorithm maintains the position placeholders while ignoring non-matching events, the task monitor can monitor tasks in the midst of non-matching events and *unspecified* activities. For example, in the `Manage Schedule` task, the user could save the email at any time – thus generating a series of non-matching events – without disrupting the task monitor.

The algorithm compares incoming events against the specified events that could possibly occur next. In cases where an event matches multiple subsequent events, an ambiguity arises. We handle this through one of many possible solutions, namely by matching the event to the placeholder that was created first chronologically.

As a user transitions among specified tasks, the task monitor builds a model of the user's task execution. The model is a graph where the nodes represent tasks and events and directed edges represent transitions between the events. Initially, the graph represents the task structure of the corresponding specification, built recursively from the specification itself. There is one graph for each specification supplied. Note that control tags are not included in the graph, since by this point, the task monitor has already decided that the generated event matches a specification.

When an event occurs, a directed edge is added (just the first time) from the last event's node to the generated event's node and the frequency and timing information is recorded. For example, for the specification in Fig. 2, if the last event was `openAppItem` and the next event is `changeAppItemProp`, the system adds a directed edge between the event's nodes, sets the transition probability to 1, and records the time between events in the last event's node. Next time, for example, if a `closeMailItem` event occurs, the system adds an edge from the `openAppItem` node to the `closeMailItem` node, updates the transition frequencies on the outgoing links to 0.5, and updates the timing information, e.g., by computing and storing the average of the values, and so on. From the model, the monitor can infer the time and transition to the next event (task) and further event (task) sequences. The tasks containing an event can always be identified from the model. The more repetitive the tasks and events are, the more accurate the inferences.

Forecasting would provide a much needed service for attention-aware systems that seek to defer delivery of information until a user reaches an opportune moment such as a boundary in their task sequence. To make an effective decision, the system must know the likelihood that the user will reach different levels of boundaries (with the assumption that interrupting at a higher-level boundary would cause less disruption) and how long it will take for the user to reach those moments. We are currently extending our implementation to provide this support.

4.5. Implementation

We engineered our system to enable others to leverage and build upon our implementation effort as much as possible. Since XML is a powerful, well-known language toolkit, it was used to construct our task description language. We believe the resulting syntactic representation is easy to use and learn. Also, there are many freely available tools for building and parsing XML documents, so our language can be quickly extended, e.g., to add new attributes to existing tags.

A key design decision was the use of a client/server architecture with multiple connection points. Any instrumented client application can connect to our event handler, register an event dictionary, and begin sending events. While the application must communicate using a defined protocol, the protocol is documented in the source distribution and is relatively simple in that it exchanges plain text XML structures. Once an application is connected, the PETDL tool can be immediately used to view the live stream of events from the application as well as to create related specifications. Once specifications are created, the task monitor can observe those tasks.

Similarly, any user-level service can connect to the task monitor and request notifications of when a user starts or finishes a specified task or subtask. Requests and notifications are exchanged using plain text XML structures as a communication protocol. Because user task recognition and monitoring is an essential component of many intelligent systems including intelligent tutoring systems (Cheikes et al., 1998), software agents (Maes, 1994), and attention aware systems (Horvitz & Apacible, 2003), our architecture reduces future implementation efforts by abstracting these commonly needed services into a single monitoring component. Our future implementation effort will be to allow user-level services to store their own information into the execution model maintained by the task monitor and to support forecasting of a user's task execution sequences.

The task monitor and the event handler were written in Python and consist of several thousand lines of programming code. Python's xml.dom library was used to validate task specifications. PETDL Maker was written in Visual Basic.NET and consists of about 2500 lines of code. Plugins for MS Outlook and MS Word were written with Visual Basic 6 and monitored events published by their respective scripting APIs. The plugin for Firefox was written in ECMAScript. We implemented our framework on an MS Windows platform because it was readily accessible to us and it was the best fit with the programming expertise of our research team.

5. Evaluation

We conducted two user studies to evaluate how well our existing implementation satisfied our design goals and to identify areas and methods for improvement. The studies were designed to represent a common scenario of use in user task modeling. The first study was conducted to learn how effective our language was for creating task specifications and how useful our graphical tool was for constructing specifications from collected events. In this study, two authors analyzed interaction videos of four users performing the same three tasks. From these observations and, with the help our graphical tool, they created specifications of the tasks.

The second study was conducted to measure how well those specifications could match the event streams generated by a different group of users performing the same tasks. Also,

the event streams collected from this study were fed into the task monitor to determine the robustness of its event matching algorithm. Together, these studies enabled us to measure length and complexity of the specifications, how many and which PETDL tags were used, and the effectiveness of the matching algorithm in the task monitor.

5.1. Users and tasks

Two authors (both male) and four users (one female) participated in the first study, and eight different users (four female) participated in the second study. The two authors were computer science graduate students familiar with regular expressions and grammars. The users consisted of undergraduate and graduate students who were experienced users of email, word processing, and web browsing software. In both studies, each user performed the same three tasks.

To keep users focused on their task, and to assure an uninterrupted experimental trial, users were given instructions on how to perform the tasks, and were then left unattended while they performed them. Details in the instructions were kept to a minimum so as to not influence the pattern of user behavior. Though individual tasks were conducted without a broader context, we constructed the tasks to involve multiple applications and placed as few restrictions as possible on user behavior.

Three tasks; document editing, Web posting, and calendar scheduling, were developed for the studies. For the document editing task, the user first located an email message in Outlook's inbox, opened the attached document in MS Word, made corrections, saved the modified document, and sent a reply from MS Outlook with the modified document attached. The document was annotated with instructions on how to correct each error.

In the Web posting task, the user navigated to a website using Mozilla Firefox, found a specific web log entry and posted a comment. The user interacted with the site to ensure that the post was anonymous, to preview the comment, and to make a given change to the comment before making the final post. Again user interaction within the task was unconstrained, with users instructed only broadly in terms of the main task goals.

The third task was a scheduling task where the user opened an email in Microsoft Outlook and scheduled an initial appointment using Outlook's calendar based on requirements in the email. The user then opened a second email and again scheduled an appointment. Because the requirements (date, time) of the appointment were ambiguous, a user may have had to re-schedule the first appointment in order to properly schedule the second appointment.

The average time to perform each task was just over 4 min, more than enough time to generate a meaningful stream of events. We selected these three tasks because they would provide a sufficient initial test of our system in the desktop domain; they involved multiple applications, are representative of tasks that users often perform, and users could perform them in a manner unconstrained enough to test the expressiveness of our task description language. This last aspect also provides a good test of how well our task monitor could handle variance in task execution.

5.2. Procedure

In both studies, a user performed practice trials of the tasks prior to the experimental tasks. After any clarifications or questions were answered, the user performed the

experimental tasks. Our plugins for the applications involved in the tasks intercepted user events and sent them to the event handler for logging. This would allow us to match the event streams to the specifications by hand and understand where and why mismatches occurred. Commercially available software was used to electronically record a user's screen interaction. Each study lasted about 30 min.

6. Measurements and results

6.1. User study I

In our first study, the two authors reviewed the event logs and screen interaction videos to create PETDL specifications for each task. Part of the final specification for the calendaring task is shown in Fig. 2 and the full specification for the document editing task is in Fig. 7. The process was iterative, with authors creating the first draft of the specifications after viewing one user's interaction, then revising that specification based on the task executions of the remaining users. Once complete, the resulting specifications expressed all of the users' task sequences.

The total time spent constructing the task specifications was not significantly more than the time needed to review the interaction videos, which was about 2 h. After specifications were developed, we counted the frequency of tags used to describe each task, summarized in Table 2. Specifications required only a small number of tags to express the tasks overall and were relatively short in length. On average, counting just the event and control tags, document editing was 14 lines, Web posting was 15 lines, and calendar scheduling was 12 lines. This shows that our description language can express event streams for practical tasks in a concise manner.

6.2. User study II

For the second study, we wanted to determine how well the specifications developed in the first study would express the task execution of a different set of users performing the same tasks. After following a procedure similar to that outlined above, we compared the new set of user event streams to the specifications. For each event, we classified the outcome as a match, a user error, or a specification error. A match meant that the specification correctly described the event. A user error occurred when the specification could express the event stream, but an error arose due to the user not performing the task as requested. For example, during the Web posting task, one user posted to the wrong web log. A specification error was when the user performed the task as requested, but the specification did not express the event stream. This was the most serious type of error.

The matching outcomes are depicted in Fig. 8. Though accuracy was task dependent, the specifications were able to reasonably express the event streams from this set of users. An inspection of the event streams showed that improved accuracy could be achieved by performing additional iterations on the task specifications; there were a number of informative events that could be added to future iterations of the task specifications. However, because this was the first time that the authors in our study had ever created task specifications or used our description language to do so, we find these results encouraging and believe that they show that creating task specifications is feasible in practice.

```
<taxonomy name="DocEdit">
  <activity name="document editing">
    <task name="manage email">
      <event name="objMailItem_Open" />
      <event name="objMailItem_AttachmentRead" />
        <task name="edit document">
          <oneOrMore>
            <choice>
                <event name="objDocument_Change" />
                <event name="objDocument_Spellcheck" />
                <event name="objDocument_Grammarcheck" />
            </choice>
          </oneOrMore>
          <event name="objDocument_Save">
              <parameter>
                <name>location</name>
                <value>desktop</value>
              </parameter>
          </event>
          <optional>
                <event name="objDocument_Close" />
          </optional>
        </task>
        <task name="reply to email">
          <event name="objMailItem_Reply" />
          <event name="objMailItem_Open" />
          <event name="objMailItem_AttachmentAdd" />
          <event name="objMailItem_Send" />
        </task>
    </task>
  </activity>
</taxonomy>
```

Fig. 7. The final task specification for the document editing task built from the first user study.

Table 2
Tag usage for specifications from the first user study

PETDL tag	Doc edit	Web search	Scheduling	Total
Task	4	5	3	9
inOrder	0	0	2	2
anyOrder	0	1	1	2
Optional	1	0	1	2
oneOrMore	1	0	1	2
zeroOrMore	1	0	0	0
repeatExactly	0	0	0	0
Choice	1	1	0	2
Event	10	9	7	24
Total tags	17	16	15	43

Because matching the event streams to the specifications was done by hand in order to categorize and understand matching errors, the next step was to test the computational matching algorithm in the task monitor. The specifications (without modification) were

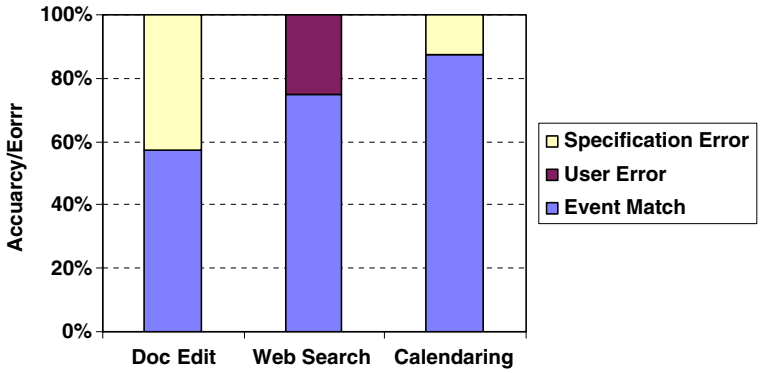

Fig. 8. Results for task recognition.

loaded into the task monitor and events from each captured stream were delivered one at a time, just as if they had come from the event handler in a live environment. For each stream, the task monitor correctly matched events to the specifications and correctly ignored events that did not match. Though the matching algorithm worked correctly, an important point is that how well a specification matches an event stream really depends on the accuracy of the specification itself, not necessarily on the matching algorithm.

Next, to simulate multiple tasks and multiple instances of the same tasks being performed, we interleaved the streams arbitrarily and sent events from this newly formed stream to the task monitor. Through inspection of the matching log, we found that the algorithm correctly matched each event to the appropriate specification and correctly ignored non-matching events. This validates the correctness of the algorithm and shows that use of the position placeholders supports multiple active tasks and multiple instances of the same tasks.

7. Discussion

7.1. Meeting the design goals

We discuss how the existing implementation of our framework has heretofore met our system design goals. To enable low-investment creation of task specifications, we developed an XML-based language that has a small number of control tags that can be used to concisely express many variations of task execution sequences. This results in specifications that are reasonably easy to read and understand and that are of relatively short length. The language is accompanied by an effective graphical tool that facilitates a macro-style authoring of specifications. The tool enables authors to construct specifications by composing control tags and events from the live user event stream and event dictionaries within an editable tree structure.

To enable tasks to be specified at multiple levels of detail, we leverage the hierarchical nesting syntax of XML. To enable specification of tasks that involve multiple applications, the language allows the names of events to be prefaced by the name or instance of the application, eliminating ambiguity among events with the same name. To accurately monitor specified tasks in the midst of unspecified activities, our monitoring algorithm uses a

list of position placeholders to mark where a user is relative to each instance of a task. This enables the monitor to follow a user's progress through multiple active tasks and multiple instances of the same task. To support forecasting of a user's task execution, we have laid the implementation groundwork by recording transition frequencies in a persistent model of task execution.

7.2. Lessons about task specification

From our experience using the system, we also learned lessons about how to better specify tasks. First, tasks should not end with control tags that may optionally occur. For example, if ⟨zeroOrMore⟩ or ⟨oneOrMore⟩ are used as the last control tag in a task, the task monitor does not know whether or not to keep waiting for more repetitions or to mark the task as complete. Second, task authors can help disambiguate task specifications. For example, some tasks can be described with multiple specifications that match the same event sequences. The same problem can be found in regular expressions like `(the)|(this)` which is equivalent to `th((e)|(is))`. While this normally does not influence regular expressions, for task specifications it causes ambiguity when matching events. To help overcome this ambiguity, it proved best to group together the longest sequence of events possible in each part of a specification. Another solution could be to review past events and consider future transition probabilities.

Our language enables an author to specify tasks at multiple levels of detail. Consistent with lessons from task modeling (Card et al., 1983), our experience is that specifying tasks to progressively finer levels of detail is progressively more difficult since many more execution sequences become possible. More experience with creating specifications for practical tasks is necessary to understand and recommend an appropriate level of detail.

7.3. Limitations and implementation issues

A limitation of our approach is that, regardless of the effectiveness of our language and tools, creating specifications will always require some amount of effort. However, we believe that the effort to create task specifications will be outweighed by the benefit of their use, e.g., mitigating the negative impact of interruption, in both safety critical and office environments. In safety critical environments, where human operators of complex systems can often be interrupted when performing critical tasks, mitigating the negative impact of interruption on task performance, error rate, and decision-making could save lives and prevent catastrophic accidents (McFarlane & Latorella, 2002). In office environments, where task performance is important, but perhaps less critical, large reductions in the frustration, annoyance, and anxiety that users too often experience due to ill-timed interruptions would also yield a meaningful benefit.

It is important to understand that specifications for tasks only need to be created *once* and can then be shared and reused. We envision specifications being produced as part of the interface design process and packaged with applications. Also, we envision a community of authors (designers, developers, IT staff, end users, etc.) willing to create and share specifications for common task environments such as for email and instant messaging, graphic design, document editing, and Web browsing. If effective user-level services can be developed and deployed, this will provide further impetus for creating and sharing specifications. In particular, we believe that developing an attention aware system – facilitated

by the use of our framework – that mitigates the negative impact of interruption would provide such a compelling service.

Another practical limitation is that the use of our framework requires applications to be properly instrumented such that the task monitor can access the user event stream. There are at least two methods to provide this instrumentation. One method is for developers of software applications to provide the necessary instrumentation during development. This is not unrealistic, as many applications developed for the Mac OS have such instrumentation, presumably due to the long availability of its system-wide scripting and event framework. The software infrastructure needed to script applications, which would provide much of the instrumentation needed to leverage our framework, is becoming increasingly available in other operating systems as well. The disadvantage of this method, however, is that instrumentation must occur per application.

An alternative is to adapt the underlying user interface management system to make the event stream accessible without modification to the applications that use it, e.g., by modifying the dynamic interface libraries loaded at startup. This method has been successfully used in several projects, such as those discussed in (Cheikes et al., 1998; Ritter et al., 2000). The advantage is that instrumentation only needs to occur once while the disadvantage is that events are lower-level and more difficult to interpret (e.g., mapping an event to the corresponding interface control). We believe that advances in directions of both methods will make the user event stream more readily available in the future.

It should also be noted that the level of detail in the instrumentation itself affects the level of detail possible in the task specifications. For example, if events related to text selection in MS Word are not made available, then authors cannot include those events in a specification. Thus, developers should instrument applications to a fine level of detail and allow authors to choose the desired level of detail when creating specifications.

When a user reaches an opportune moment in a task sequence, an attention aware system must be able to respond quickly, presumably on the order of a few hundred milliseconds, else the window of opportunity for interruption may be lost. This is challenging, as once an application sends the triggering event, the task monitor must, at a minimum, capture, match, and handle the event and then notify the system. Whether this process would allow the system to deliver the information in time depends on several factors such as the computational speed of the machine executing the task monitor, the efficiency of its algorithms, and the size of the temporal window around the opportune moment. Empirical research is needed to provide numerical estimates for the latter factor.

8. Future work

Our future work seeks to implement algorithms for forecasting a user's task execution sequences based on historical observations of task execution, extend our existing plugins to further instrument the applications, develop plugins for additional, commonly used applications, and develop an attention aware system that uses our framework to defer the delivery of peripheral information until selected boundary points in a user's task sequence.

Once developed, such a system must undergo extensive evaluation to demonstrate its utility for users in realistic settings. An evaluation should focus on at least two elements. First, an evaluation must measure how well the task monitor can forecast a user's task sequences for commonly used applications, as accurate forecasting would be critical for

deciding when to interrupt. This also implies that task specifications must effectively describe common patterns of application use and that the task monitor can accurately recognize those patterns.

Second, since the system would defer delivery of information until boundary points during task execution, the evaluation must investigate acceptable tradeoffs between decreased awareness and increased mitigation of disruption. Though results from empirical studies show that decreased awareness can indeed be exchanged for increased mitigation of disruption (Bailey & Konstan, 2005), the tradeoffs that would be necessary for users to adopt such systems in practice needs to be better understood.

9. Conclusion

Existing systems that reason about when to interrupt do not have access to task models that would allow for finer-grained temporal reasoning. To enable this reasoning, we have presented an integrated framework for specifying and monitoring user tasks. For task specification, our framework provides a language that supports expressive specification of tasks using a concise notation. For task monitoring, our framework provides an event database and handler that manage events from any instrumented application and a task monitor that observes a user's progress through specified tasks. Results were also presented from two user studies showing that our language can be used to effectively specify practical tasks and that our monitoring system can accurately follow a user's progress. Our framework facilitates instrumentation of office environments as well as safety critical domains to provide compelling user-level services such as intelligent tutoring, context-sensitive help, and intelligent interruption management.

Acknowledgements

We would like to thank the anonymous reviewers for constructive comments on an earlier draft of this article. This work was supported in part by a grant from the National Science Foundation under award no. IIS 05-34462.

References

Adamczyk, P. D., & Bailey, B. P. (2004). If not now when? The effects of interruptions at various moments within task execution. *Proceedings of the ACM conference on human factors in computing systems*, pp. 271–278.

Albrecht, D., Zukerman, I., Nicholson, A., & Bud., A. (1997). Towards a bayesian model for keyhole plan recognition in large domains. *Proceedings of user modeling*, pp. 365–376.

AppleScript. (Apple Computer). Available from http://www.apple.com/applescript, February 15, 2005.

Bailey, B. P., & Konstan, J. A. (2005). On the need for attention aware systems: Measuring effects of interruption on task performance, error rate, and affective state. *Journal of Computers in Human Behavior, special issue on attention aware systems*.

Bailey, B. P., Konstan, J. A., & Carlis, J. V., (2001). The effects of interruptions on task performance, annoyance, and anxiety in the user interface. In *Proceedings of the IFIP TC.13 international conference on human–computer interaction*, Tokyo, Japan, pp. 593–601.

Byrne, M. D., & Anderson, J. R. (1998). Perception and action. In J. R. Anderson & C. Lebiere (Eds.), *The atomic components of thought*. Mahwah, NJ: Lawrence Erlbaum Associates, Inc.

Byrne, M. D., Wood, S. D., Sukaviriya, P., Foley, J. D., & Kieras, D. E. (1994). Automating interface evaluation. In *Proceedings of the ACM conference on human factors in computing systems*, Boston, MA, pp. 232–237.

Carberry, S. (2001). Techniques for plan recognition. *User Modeling and User-Adapted Interaction, 11*, 31–48.

Card, S., Moran, T., & Newell, A. (1983). *The psychology of human–computer interaction*. Mahwah, NJ: Lawrence Erlbaum Associates, Inc.

Carr, D. A. (1994). Specification of interface interaction objects. In *Proceedings of the ACM conference on human factors in computing systems*, Boston, MA, pp. 372–378.

Cheikes, B. A., Geier, M., Hyland, R., Linton, F., Rodi, L., & Schaefer, H.-P. (1998). Embedded training for complex information systems. *International Journal of Artificial Intelligence in Education*, 314–334.

Cutrell, E., Czerwinski, M., & Horvitz, E. (2001). Notification, disruption and memory: Effects of messaging interruptions on memory and performance. In *Proceedings of the IFIP TC.13 international conference on human–computer interaction*, Tokyo, Japan, pp. 263–269.

Czerwinski, M., Cutrell, E., & Horvitz, E. (2000a). Instant messaging and interruption: Influence of task type on performance. In *OZCHI 2000 conference proceedings*, Sydney, Australia, pp. 356–361.

Czerwinski, M., Cutrell, E., & Horvitz, E. (2000b). Instant messaging: Effects of relevance and timing. In *People and Computers XIV: Proceedings of HCI*, pp. 71–76.

Czerwinski, M., Horvitz, E., & Wilhite, S. (2004). A diary study of task switching and interruptions. *Proceedings of the ACM conference on human factors in computing systems*, pp. 175–182.

Franklin, D., Budzik, J., & Hammond, K. (2002). Plan-based interfaces: Keeping track of user tasks and acting to cooperate. In *International conference on intelligent user interfaces*, pp. 79–86.

Gong, R., & Kieras, D. (1994). A validation of the goms model methodology in the development of a specialized, commercial software application. In *Proceedings of the ACM conference on human factors in computing systems*, Boston, MA, pp. 351–357.

Guttag, J., & Horning, J. J. (1980). Formal specification as a design tool. *Proceedings of the seventh symposium on programming languages*, pp. 251–261.

Harel, D. (1987). Statecharts: A visual formulation for complex systems. *Science of Computer Programming, 8*(3), 231–274.

Hartson, H. R., Siochi, A. C., & Hix, D. (1990). The uan: A user-oriented representation for direct manipulation interface designs. *ACM Transactions on Information Systems, 8*(3), 181–203.

Horvitz, E. (1999). Principles of mixed-initiative user interfaces. In *Proceedings of the ACM conference on human factors in computing systems*, pp. 159–166.

Horvitz, E., & Apacible, J. (2003). Learning and reasoning about interruption. *Proceedings of the fifth ACM international conference on multimodal interfaces*, pp. 20–27.

Horvitz, E., Breese, J., Heckerman, D., Hovel, D., & Rommelse, K. (1998). The lumiere project: Bayesian user modeling for inferring the goals and needs of software users. *Proceedings of the 14th conference on uncertainty in artificial intelligence*, pp. 256–265.

Horvitz, E., Jacobs, A., & Hovel, D. (1999). Attention-sensitive alerting. In *Conference proceedings on uncertainty in artificial intelligence*, pp. 305–313.

Hudson, S. E., Fogarty, J., Atkeson, C. G., Avrahami, D., Forlizzi, J., & Kiesler, S., et al. (2003). Predicting human interruptibility with sensors: A wizard of oz feasibility study. *Proceedings of the ACM conference on human factors in computing systems*, pp. 257–264.

Iqbal, S. T., Adamczyk, P. D., Zheng, S., & Bailey, B. P. (2005). Towards an index of opportunity: Understanding changes in mental workload during task execution. In *Proceedings of the ACM conference on human factors in computing systems*, pp. 311–320.

Iqbal, S. T., Zheng, X. S., & Bailey, B. P. (2004). Task evoked pupillary response to mental workload in human–computer interaction. *Proceedings of the ACM conference on human factors in computing systems*, (short paper).

John, B. E. (1995). Why goms? *Interactions, 2*, 80–89.

John, B. E., & Kieras, D. E. (1996). The goms family of user interface analysis techniques: Comparison and contrast. *ACM Transactions on Computer–Human Interaction, 3*(4), 320–351.

John, B. E., Vera, A., Matessa, M., Freed, M., & Remington, R. (2002). Automating cpm-goms. In *Proceedings of the ACM conference on human factors in computing systems*, Minneapolis, MN, pp. 147–154.

Kieras, D. E., & Meyer, D. E. (1997). An overview of the epic architecture for cognition and performance with application to human–computer interaction. *Human–Computer Interaction, 12*, 391–438.

Kieras, D. E., Wood, S. D., Abotel, K., & Hornof, A. (1995). Glean: A computer-based tool for rapid goms model usability evaluation of user interface designs. *Proceedings of the ACM symposium on user interface software and technology*, pp. 91–100.

Latorella, K. A. (1998). Effects of modality on interrupted flight deck performance: Implications for data link. In *42nd Annual meeting of the human factors and ergonomics society*, pp. 87–91.

Lieberman, H. (1997). Autonomous interface agents. In *Proceedings of the ACM conference on human factors in computing systems*, pp. 67–74.

Maes, P. (1994). Agents that reduce work and information overload. *Communications of the ACM, 37*(7), 30–40.

Maglio, P., Barrett, R., Campbell, C. S., & Selker, T. (2000). Suitor: An attentive information system. In *Proceedings of the international conference on intelligent user interfaces*, pp. 169–176.

Maulsby, D. (1997). Inductive task modeling for user interface customization. In *Proceedings of the international conference on intelligent user interfaces*, pp. 233–236.

McFarlane, D. C. (1999). Coordinating the interruption of people in human–computer interaction. In *Proceedings of the IFIP TC.13 international conference on human-computer interaction*, pp. 295–303.

McFarlane, D. C., & Latorella, K. A. (2002). The scope and importance of human interruption in hci design. *Human–Computer Interaction, 17*(1), 1–61.

Miyata, Y., & Norman, D. A. (1986). The control of multiple activities. In D. A. Norman & S. W. Draper (Eds.), *User centered system design: New perspectives on human-computer interaction*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Monk, C. A., Boehm-Davis, D. A., & Trafton, J. G. (2002). The attentional costs of interrupting task performance at various stages. In *Proceedings of the human factors and ergonomics society 46th annual meeting*.

Rich, C., & Sidner, C. L. (1998). Collagen: A collaboration manager for software interface agents. *User Modeling and User-Adapted Interaction, 8*(3/4), 315–350.

Ritter, F. E., Baxter, G. D., Jones, G., & Young, R. M. (2000). Supporting cognitive models as users. *ACM Transactions on Computer–Human Interaction, 7*(2), 141–173.

Shneiderman, B. (1982). Multiparty grammars and related features for defining interactive systems. *IEEE Transactions on Systems, Man, and Cybernetics, 12*(2), 148–154.

Siochi, A. C., & Hartson, H. R. (1989). Task-oriented representation of asynchronous user interfaces. *Proceedings of the ACM conference on human factors in computing systems*, pp. 183–188.

Speier, C., Valacich, J. S., & Vessey, I. (1999). The influence of task interruption on individual decision making: An information overload perspective. *Decision Sciences, 30*(2), 337–360.

St. Amant, R., & Riedl, M. O. (2001). A perception/action substrate for cognitive modeling in hci. *International Journal of Human–Computer Studies, 55*, 15–39.

Szekely, P., Luo, P., & Neches, R. (1993). Beyond interface builders: Model-based interface tools. In *Proceedings of INTERCHI*, pp. 383–390.

Tauber, M. J. (1990). Etag: Extended task action grammar. A language for the description of the user's task language. In *INTERACT*, 27–31 August, 1990, Cambridge, UK, pp. 163–168.

Zacks, J., Tversky, B., & Iyer, G. (2001). Perceiving, remembering, and communicating structure in events. *Journal of Experimental Psychology: General, 130*(1), 29–58.

Zijlstra, F. R. H., Roe, R. A., Leonora, A. B., & Krediet, I. (1999). Temporal factors in mental work: Effects of interrupted activities. *Journal of Occupational and Organizational Psychology, 72*, 163–185.

Brian P. Bailey is an Assistant Professor in the Department of Computer Science at the University of Illinois-Urbana. His research investigates developing interactive design tools that better support human creativity, user interfaces for pervasive computing, computational systems that manage human attention, and other areas of human–computer interaction. Dr. Bailey received his Ph.D. from the University of Minnesota, Minneapolis, in 2002. His is a member of the ACM and the current editor of the ACM SIGCHI Bulletin.

Piotr D. Adamczyk is a graduate student in Human Factors and Library and Information Science at the University of Illinois-Urbana. He received a B.S. in Mathematics and Computer Science from the University of Illinois in 2003. His research interests include applied attention and social computing. He is a member of the ACM.

Tony Y. Chang is a software engineer working for Google.com. His interests are in rich web applications and collaborative software. He received his M.S. in Computer Science from the University of Illinois-Urbana in 2004. He is a member of the ACM.

Neil A. Chilson is currently a student at The George Washington University Law School with interests in intellectual property, patent, and privacy law. He received his M.S. in Computer Science from the University of Illinois-Urbana in 2005 and his B.S. degree in Computer Science from Harding University in 1999.