Planning for Desktop Services

Ronald P. A. Petrick
School of Informatics
University of Edinburgh
Edinburgh EH8 9LW, Scotland, UK
rpetrick@inf.ed.ac.uk

Abstract

A user's interaction with a computer operating system is most commonly reflected in the use of "desktop" application programs. In this paper we investigate the prospect of building plans that link together the services provided by such applications using an inter-process communication language called DCOP (Desktop COmmunication Protocol). Such services can be used to manipulate or query desktop applications, often in a manner similar to the standard user interfaces of those applications, while offering the possibility of a practical interface that a planning agent can utilize in a real software setting. Using the knowledge-level conditional planner PKS, we show how we can construct plans for controlling a set of existing desktop applications in the open source K Desktop Environment (KDE), and illustrate our approach with a series of fully executable examples that include application control and information gathering, under conditions of incomplete information and sensing.

Introduction

In an operating system environment, a user's experience is typically reflected in the use of *desktop* applications, programs such as e-mail clients, word processors, media players, and web browsers that provide the user with an abstracted interface to the tasks that can be performed with a modern operating system. From the point of view of an agent (human or artificial) operating in such an environment, application programs provide *services* that can be exploited by the agent to fulfil its goals or objectives. Designing an artificial agent that can use such services effectively in a real software setting, however, is both a potentially beneficial and particularly challenging task.

Planning in an operating system environment is not a new idea. For instance, the Softbot project (Etzioni *et al.* 1993; Etzioni & Weld 1994) focuses on the design of software agents capable of functioning in the UNIX and Internet environments, where software actions such as *ftp* and *lpr* are used as effectors for manipulating the environment, and actions like *ls* and *finger* are available as sensors for gathering information from the environment. The knowledge-level planning work of (Petrick & Bacchus 2002) similarly uses UNIX as a domain for planning with incomplete information and sensing actions. The extensive work on planning for web services, and in particular the web service composition problem (e.g., (Pistore *et al.* 2005; McIlraith & Son 2002; Martínez & Lespérance 2004), among others), continues to investigate how programs or devices available through the World Wide Web can be used to achieve an agent's goals.

In this paper we focus on the problem of constructing a planning agent that is capable of interacting with a set of common application programs in a desktop environment. Unlike those approaches that focus on modelling low-level UNIX commands, the desktop applications we are interested in typically operate at a much higher level of abstraction. For instance, a media player may work with playlists constructed from meta-level properties like user preference, artist, and genre, compared with programs like *ls* and *lpr* that directly manipulate files and directories. Furthermore, since our focus is on the desktop environment, we are more concerned with "local" services rather than external web services, except for those desktop applications that provide immediate interfaces to the external world (e.g., web browsers, media players that can play remote audio streams, etc.). Finally, since we are interested in constructing *practical* planning agents, we will focus on real applications running in a real desktop environment, and use existing inter-application communication facilities as much as possible, to take advantage of technology already in use.

To this end, we will work within the K Desktop Environment (KDE),[1] an open source, freely available desktop for UNIX-like systems. One of the novel features of this platform is its rich inter-process communication language DCOP (Desktop COmmunication Protocol), which provides the infrastructure for applications to publish their services to other programs running in the environment. What makes this interface particularly interesting is that an application's published DCOP services are often quite similar to the functions a human user can perform through the application's standard (usually graphical) interface. It is this high level of abstraction that we hope to leverage for planning purposes.

To generate plans in this setting we will work with the knowledge-level conditional planner PKS (Petrick & Bacchus 2002) and makes use of PKS's ability to reason about incomplete information using limited, but expressive, representations of the agent's knowledge state. While previous work has applied PKS to both the problem of planning with

[1] The KDE project website is located at www.kde.org.

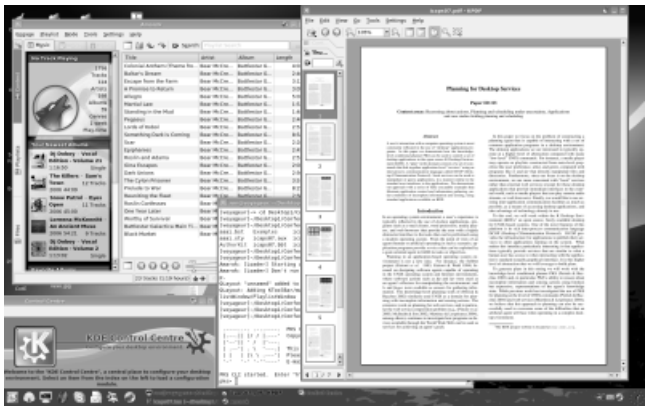

Figure 1: The K Desktop Environment (KDE)

UNIX commands (Petrick & Bacchus 2004) and web services (Martínez & Lespérance 2004), we also believe that this approach can also be successful in the desktop environment described above.

This paper is organized as follows. First, we will describe DCOP and its use in KDE-based applications. Second, we will briefly describe the operation of PKS. Third, we will describe a series of planning examples in our desktop domain that are fully executable using PKS and existing KDE applications, and briefly mention some details concerning plan execution. Finally, we will discuss some issues related to our approach and describe the future directions of this research. Our goal in this paper is twofold: to highlight the desktop services environment as a real-world (software) domain that has been largely overlooked as a testbed for applying planning technology, and to provide results of our preliminary experiments in this domain.

Desktop COmmunication Protocol (DCOP)

The Desktop COmmunication Protocol (DCOP) (Brown *et al.* 2003) is a protocol that enables inter-process communication between applications running in the K Desktop Environment (see Figure 1). DCOP allows KDE-based applications to publish some of their capabilities or *services* for interoperability with other applications, by providing a public interface to these features. In particular, developers are free to decide what services their applications should make available, with DCOP supplying the common interface to these services. What makes DCOP different from other IPC languages (e.g., CORBA, ICE, etc.) is that underlying support for the protocol is provided by KDE itself, where DCOP forms an integral component of the desktop environment.

Although any feature of an application can be exported as a DCOP service, in practice, many applications provide services that are similar to those that can be performed by a user working with the standard (usually graphical) interface to the application. Since DCOP operates on particular instances of running applications, these services typically include commands for changing or querying an application's state. For example, Figure 2 shows a fragment of the DCOP interface for the Amarok media player. Functions like *play*

```
void mute()
void pause()
void play()
void setVolume(int volume)
void enableRepeatTrack(bool enable)
QString artist()
QString nowPlaying()
QString title()
...     ...
```

Figure 2: A fragment of a media player's DCOP interface

and *setVolume* provide a means of changing the state of the application, while *nowPlaying* and *title* provide feedback about the application and are more akin to information gathering operations. Moreover, these services provide a very abstract (and natural) interface to the application.

From a development point of view, the DCOP interface is available in a variety of programming languages, including C++, C, Perl, and Python. In addition, DCOP services can also be accessed through a command line interface provided by KDE, making them useful for writing scripts (Wheeler 2003). For instance,

```
$ dcop amarok player play
```

directs Amarok to play its current playlist using a DCOP service available in the `player` category.[2]

In this paper we are interested in using DCOP as an action-level language for planning, where the plans we generate will contain operations that closely correspond to DCOP services available in existing KDE applications.[3] Thus, our plans will be similar to DCOP scripts, but with extra plan-level control directives. The prospect of planning at the DCOP level is particularly appealing due to the level of abstraction already built into many of the available services. Building a formal action model for a planner that uses alternate interfaces to the same application (e.g., UNIX-style command-line options) may mean describing such actions in much lower-level detail. As well, alternative interfaces to the same set of features may not be available for many applications, except through standard graphical interfaces. Finally, the variety of programming languages available for accessing the DCOP interface leaves us a great deal of flexibility for executing plans.

Planning with Knowledge and Sensing (PKS)

To generate plans in this setting we will use PKS (Planning with Knowledge and Sensing), a conditional planner that can construct plans in the presence of incomplete information and sensing actions (Petrick & Bacchus 2002; 2004). PKS takes a "knowledge-level" approach to plan generation by reasoning about its own knowledge and how its knowledge state—rather than the world state—changes due to action. PKS also works with a restricted subset of a first-order language, and a limited amount of inference in

[2] Services are typically categorized so that an application/category/name triple is need to identify a particular service.

[3] All the examples we describe in this paper use actual KDE applications and DCOP interfaces, unless otherwise noted.

that subset, which allows it to support a rich representation that includes non-propositional features such as functions and variables, and to reason efficiently with that representation. This approach differs from planners that work with propositional representations over which complete reasoning is feasible, or approaches that model incomplete knowledge based on sets of possible worlds (e.g., BDDs (Bryant 1992), Graphplan-like structures (Weld, Anderson, & Smith 1998), clausal representations, or other such techniques). By working at the knowledge level, PKS can often abstract its reasoning away from irrelevant distinctions that occur at the world level.

PKS is based on a generalization of STRIPS (Fikes & Nilsson 1971). In STRIPS, the state of the world is modelled by a single database; actions update this database and, by doing so, update the planner's world model. In PKS, the planner's knowledge state, rather than the world state, is represented by a set of five databases, the contents of which have a fixed, formal interpretation in a modal logic of knowledge. Actions can modify any of the databases, which has the effect of updating the planner's knowledge state. To ensure efficient inference, PKS restricts the type of knowledge (especially disjunctions) that it can represent in each database. We briefly mention these databases below.

$\mathbf{K_f}$: This database is similar to a standard STRIPS database except that both positive and negative facts are permitted and the closed world assumption is not applied. K_f is used for modelling the effects of actions that change the world. K_f can include any ground literal ℓ, where $\ell \in K_f$ means "the planner knows ℓ." K_f can also contain information about known function mappings.

$\mathbf{K_w}$: This database models the plan-time effects of "binary" sensing actions. $\phi \in K_w$ means that at plan time the planner either "knows ϕ or knows $\neg\phi$," and that at execution time this disjunction will be resolved. PKS is able to use such "know-whether" information to construct conditional plans.

$\mathbf{K_v}$: This database stores information about function values that will become known at execution time. In particular, K_v can model the plan-time effects of sensing actions that return numeric values. K_v can contain any unnested function term f, where $f \in K_v$ means that at plan time the planner "knows the value of f." At execution time the planner will have definite information about f's value. As a result, PKS is able to use K_v terms as "run-time variables" (Etzioni *et al.* 1992) in its plans.

The fourth database, K_x, models the planner's "exclusive-or" knowledge of literals, namely that the planner knows "exactly one of a set of literals is true." Such knowledge is common in many planning scenarios. The fifth database, LCW, stores the planner's "local closed world" information (Etzioni, Golden, & Weld 1994), i.e., instances where the planner has complete information about the state of the world. We will not use K_x or LCW in this paper.

PKS actions are modelled as queries and updates to the databases. Action preconditions are specified by lists of primitive queries that ask simple questions about the state of the planner's knowledge: (i) Kp, is p known to be true?, (ii) $K_v t$, is the value of t known?, (iii) $K_w p$, is p known to be true or known to be false (i.e., does the planner know-whether p?), or (iv) the negation of queries (i)–(iii). An inference algorithm evaluates queries by checking database contents, taking into consideration the interaction between different types of knowledge. Action effects are described as updates to the planner's knowledge state, and are specified by collections of STRIPS-style "add" and "delete" operations that modify the contents of the individual databases. For example, *add*$(K_f, \neg\phi)$ would add $\neg\phi$ to K_f, and *del*(K_w, ϕ) would remove ϕ from K_w. Actions are permitted to have ADL-style context-dependent effects (Pednault 1989), where the secondary preconditions of an effect are also described by lists of primitive queries. Actions and goals can also make use of a limited form of quantification that ranges over known instantiations of x (e.g., $\forall^K x$ and $\exists^K x$).

Action	Preconditions	Effects
pickup(x)	K(*handempty*)	*add*$(K_f$, *holding*$(x))$ *add*$(K_f, \neg$*handempty*$)$
inspect(x)	K(*holding*(x))	*add*$(K_w$,*fragile*$(x))$

Table 1: PKS actions

PKS constructs plans by applying actions in a simple forward-chaining manner: if the preconditions of an action are satisfied by the planner's knowledge state, then the action's effects are applied to the state to produce a new knowledge state. Planning then continues from this new state. For actions with context-dependent effects, secondary preconditions are similarly evaluated against the knowledge state to determine if their effects should be applied. PKS can also add a conditional branch to a plan provided it has K_w knowledge of a formula ϕ. Along one branch (K^+), ϕ is assumed to be known while along the other branch (K^-), $\neg\phi$ is assumed to be known. Planning continues along each branch using the new knowledge states, until each branch satisfies the goal, also specified as a list of primitive queries.

Consider the two actions in Table 1, *pickup*(x) and *inspect*(x), and consider an initial knowledge state defined by $K_f = \{$*handempty*$\}$, where all the other databases are empty. If the planner knows about an object *vase* then since *handempty* is in K_f, the preconditions of *pickup*(*vase*) would be satisfied. Applying this action results in the new state represented by the updated database $K_f = \{\neg$*handempty*, *holding*(*vase*)$\}$. At this point the preconditions to *inspect*(*vase*) are also satisfied since *holding*(*vase*) is in K_f. Applying this action leaves K_f unchanged but produces $K_w = \{$*fragile*(*vase*)$\}$, indicating that the planner knows whether the vase is fragile or not. At this point the planner could construct a conditional branch in the plan: along one branch it would assume that *fragile*(*vase*) is known, while along the other branch it would assume $\neg$*fragile*(*vase*) is known. Planning can then continue along each of these new branches.

Planning for Desktop Services

One of our aims in this paper is to demonstrate that we can use PKS to construct plans whose actions closely correspond to the DCOP services provided by common KDE applica-

Action	Precond.	Effects
amarok::playlist::		
addMedia(x)	$K(media(x))$	$add(K_f, inplaylist(x))$ $add(K_f, track(x) = total + 1)$ $add(K_f, total = total + 1)$
clearPlaylist		$add(K_f, total = 0)$ $add(K_f, current = 0)$ $\forall^K x.del(K_f, inplaylist(x))$ $\forall^K x.del(K_f, track(x))$
amarok::player::		
play		$K(total > 0) \Rightarrow$ $add(K_f, playing)$ $K(current = 0) \Rightarrow$ $add(K_f, current = 1)$
next		$K(total > current) \Rightarrow$ $add(K_f, current \mathrel{+}= 1)$

Table 2: Amarok media player actions

tions. In this section we present a series of examples that illustrate the practicality of the DCOP services provided by real KDE applications, which we feel lends support to our argument that generating plans at this level of abstraction is both feasible and desirable.

In each example we consider a set of DCOP services, provide a PKS representation of those services, and describe some simple plans we can generate using the PKS encoding. From a planning point of view, these domains present a number of interesting challenges, including reasoning about incomplete information and sensing actions, resource management, function manipulation, and arithmetic evaluation. In each example we will only model a portion of the total services available for a given application. Most importantly, we have attempted to make our domain encoding (and subsequent plans) as true to the DCOP interface as possible. While we must still postprocess our plans for execution purposes (described below), our aim is to keep this step as minimal as possible. For exposition purposes, however, we have simplified our examples in some cases, for instance using short identifiers for filenames and applications instead of fully instantiated paths.

All of our examples were generated using the latest public version of PKS (version 0.7) running on a 1.86 GHz processor with 2Gb of RAM available. All generated plans were subsequently executed on the same system using KDE 3.5.7.

Controlling an application

In the first domain we consider a set of actions for controlling Amarok, a popular KDE media player.[4] The DCOP interface provided by Amarok consists of functions for manipulating all aspects of the application, including the playlist, music collection, and player state. In this example we will only consider four DCOP services: *addMedia*(x), which adds a valid media file to Amarok's playlist; *clearPlaylist*, which removes all entries from the current playlist; *play*, which instructs the player to start playing the current playlist; and *next*, which tells the player to advance to the next track in the playlist. Table 2 lists these services and their corresponding representation as PKS actions.[5]

The PKS actions in Table 2 do not require sensing, but use PKS's ability to manipulate functions and to perform simple arithmetic reasoning. For instance, *track*(x) is a function denoting the position of track x in the playlist, *total* stores the total number of entries in the playlist, and *current* denotes the "active" track (which may or may not be playing). The predicate *inplaylist*(x) indicates x is in the playlist, and *playing* indicates whether or not a track is playing.

Using these actions was can generate a number of interesting plans that are immediately executable on KDE using the DCOP interface. We consider three simple examples. In each case the planner initially has knowledge of three media files, denoted by the initial database $K_f = \{media(track1), media(track2), media(track3)\}$, where all other databases are empty.

If we present PKS with the goal $K(playing)$, i.e., bring the application to a state where it is playing, then PKS is able to construct the plan:

amarok::playlist::clearPlaylist
amarok::playlist::addMedia(track1)
amarok::player::play

In other words, the media player can start playing provided it has first added a track to the playlist.

We can also instruct PKS to achieve the more complex goal $K(playing) \wedge K(track(track3) = current) \wedge \forall^K x.K(media(x)) \Rightarrow K(inplaylist(x))$, i.e., ensure *track3* is playing and all known tracks have been added to the playlist. Doing so produces the plan:

amarok::playlist::clearPlaylist
amarok::playlist::addMedia(track3)
amarok::playlist::addMedia(track1)
amarok::playlist::addMedia(track2)
amarok::player::play

In this case PKS achieves the goal by ensuring *track3* is the first track added to the playlist. The occurrence of *play* as the last action is completely arbitrary and PKS could alternatively build plans where *play* occurs at any point after the first *addMedia* action.

Finally, if we provide the goal $K(playing) \wedge K(total = current) \wedge \forall^K x.K(media(x)) \Rightarrow K(inplaylist(x))$, i.e., ensure all tracks are loaded and the last track is playing, then PKS generates the plan:

amarok::playlist::clearPlaylist
amarok::playlist::addMedia(track1)
amarok::playlist::addMedia(track2)
amarok::playlist::addMedia(track3)
amarok::player::play
amarok::player::next
amarok::player::next

[4]The Amarok website is located at `amarok.kde.org`.

[5]We will use the notation *app::category::service* to refer to a service's full name. Thus, *amarok::player::play* would refer to Amarok's *play* service found in the *player* service category.

Action	Precond.	Effects
amarok::player::		
isPlaying		$add(K_w, result(playing))$
isOsdEnabled		$add(K_w, result(osd))$
knotify::		
$notify(x, y)$	$K(service(x))$ $K(msgsend(x, y))$	$add(K_f, notified(?x))$
Domain specific update rules		
$K(result(x)) \Rightarrow add(K_f, msgsend(x, true))$ $K(\neg result(x)) \Rightarrow add(K_f, msgsend(x, false))$		

Table 3: Amarok information state actions

In this plan the *next* action is required to advance the actively playing track to the end of the playlist.

We note that the above examples only model some of the basic controls provided by Amarok's DCOP interface, which permits much more sophisticated access to the application. (For instance, we do not model automatic track changes, or the many services available for accessing the playlist and music collection.) Even so, these examples illustrate the high degree of control that DCOP services can provide over an application, and demonstrate their value in a practical planning setting: these services closely correspond to the actions a user could perform using the application's standard graphical interface.

Querying the state of an application

In the previous example we used a set of DCOP services to change the state of the Amarok media player. In this example we consider two DCOP services that let us access Amarok's internal state. In this case we will use such services as information gathering actions and inform the user as to the results gathered (also using a DCOP service).

The set of DCOP services we consider is listed in Table 3, along with the PKS encoding of these services. The *isPlaying* service queries whether or not the media player is currently playing, while the *isOsdEnabled* service determines if the player's on-screen display is enabled or not. These two services are modelled as PKS sensing actions that have the effect of adding information to the K_w database. The $notify(x, y)$ service is provided by an application called Knotify which, among other things, is able to display an information dialogue box to the user. We have simplified our encoding of this service for our particular purpose, where x denotes a service name and y the message to be sent. The domain specific update rules, which are automatically applied by PKS after each action application, are used as part of our wrapper around *notify*.

If we consider the scenario where PKS knows about the two Amarok services, denoted by the initial database $K_f = \{service(playing), service(osd)\}$ (where the other databases are empty), then presenting PKS with the goal of sending a notification for each known service, $\forall^K x.K(service(x)) \Rightarrow K(notified(x))$, results in the plan:

Action	Precond.	Effects
KWeatherService::WeatherService::		
$stationCode(x)$	$K_v(x)$	$add(K_w, validCode)$ $add(K_v, code)$
$temperature(x)$	$K(validCode)$ $K(code = x)$	$add(K_v, stationTemp)$
external::kdialog::		
inputbox		$add(K_v, stationName)$

Table 4: Kweather domain actions

amarok::player::isPlaying
amarok::player::isOsdEnabled
branch(result(osd))
K^+:
  branch(result(playing))
  K^+:
    knotify::notify(playing,true)
    knotify::notify(osd,true)
  K^-:
    knotify::notify(playing,false)
    knotify::notify(osd,true)
K^-:
  knotify::notify(osd,false)
  branch(result(playing))
  K^+:
    knotify::notify(playing,true)
  K^-:
    knotify::notify(playing,false)

In this case, our plan wraps PKS-level controls around DCOP-level services. The plan begins by querying Amarok's state using the two Amarok services, which provides the planner with "know whether" information about $result(playing)$ and $result(osd)$. A conditional branch is added to the plan at this point, based on $result(osd)$, letting the planner reason about the two possible outcomes of this information: along the positive branch (K^+), $result(osd)$ is assumed to be true (i.e., the on-screen display is enabled); along the negative branch (K^-), $\neg result(osd)$ is assumed to be true (i.e., the on-screen display is disabled). At this point, the update rules are applied. Along the branch where $result(osd)$ is true, $msgsend(osd, true)$ is also made true, while along the other branch, $msgsend(osd, false)$ is made true. In other words, PKS comes to know what message it should send depending on the status of the *isOsdEnabled* service. The second nested branch for $result(playing)$ performs a similar task for determining what message should be sent for the *isPlaying* service. The *notify* actions in the plan handle the four possible combinations of outcomes, one for each branch in the plan.

Desktop interfaces to web services

In the third domain we consider a desktop-based interface to a web service: the Kweather applet that can access a remote weather server. We are not concerned with the particular web service in question, but instead with the DCOP services this application offers for accessing the web service.

Table 4 lists two DCOP functions provided by Kweather.

stationCode(x) takes a name x of a weather station location and returns a station ID code, provided the name is valid. *temperature*(x) returns the temperature at a given station specified by its ID code x, provided that code is valid. The third service listed in Table 4, *inputbox*, is not a DCOP service but a function provided by the *kdialog* application which is typically used in script writing to prompt a user for input. Executing this service causes a graphical dialogue box to be shown, allowing the user to enter a string of text.

Table 4 presents a PKS encoding of the three services. (The interaction between *validCode* and station codes has been simplified and can be handled in a more general way). This representation leans heavily on PKS's ability to use functions. For instance, *code* is used to represent a station code, while *stationName* represents a station name. *validCode* is a predicate indicating whether the current *code* is valid or not. Our encoding of *stationCode*(x) also has an effect that adds *validCode* to K_w. PKS can use such information to add conditional branches to a plan (see below).

If PKS starts with an initial state where all of PKS's databases are empty, and is presented with the goal $K_v(\textit{stationTemp}) \vee K(\neg \textit{validCode})$, i.e., come to know the temperature or report failure, then it can construct the plan:

external::kdialog::inputbox
KWeatherService::WeatherService
::stationCode(stationName)
branch(validCode)
K^+:
KWeatherService::WeatherService
::temperature(stationCode)
K^-:
nil

The result is a conditional plan with a single branch. First, the plan prompts the user for input using *inputbox*. Next, it attempts to determine the station code from the user's input. The function *stationName* in this case is used as an argument to *stationCode* as a type of run-time variable that will be replaced by the actual value of *stationCode* at execution time. The branch in the plan is structured to reason about the outcome of *validCode*: along the K^+ branch *validCode* is assumed to be true, while along the K^- branch $\neg$*validCode* is assumed to be true. In the first branch, the *temperature* service is used to obtain the station's temperature, where *stationCode* acts as a run-time variable. In the other branch, the plan simply terminates.

The importance of this example lies in our use of local desktop services to access external web services. In particular, DCOP provides an abstraction layer around much of the uncertainty concerning remote services (e.g., network reliability) and supplies a common interface that lets us avoid having to manage multiple service description languages and data exchange formats. Of course, we can only take advantage of web services in this manner provided an application with a suitable DCOP interface exists (or if we build one). Since many applications blur the line between desktop and network (especially the Internet), however, finding ways to use such services effectively is a worthwhile endeavour.

Action	Precond.	Effects
external::dcop::		
find(x)	$K(\textit{KdeApp}(x))$	*add*$(K_w, \textit{running}(x))$
klauncher::		
kdeinit_exec(x)	$K(\textit{KdeApp}(x))$ $K(\neg\textit{running}(x))$	*add*$(K_f, \textit{running}(x))$
app::mainwindow::		
minimize(x)	$K(\textit{running}(x))$	*add*$(K_f, \textit{minimized}(x))$ *add*$(K_f, \neg\textit{maximized}(x))$
maximize(x)	$K(\textit{running}(x))$	*add*$(K_f, \textit{maximized}(x))$ *add*$(K_f, \neg\textit{minimized}(x))$
restore(x)	$K(\textit{running}(x))$	*add*$(K_f, \neg\textit{minimized}(x))$ *add*$(K_f, \neg\textit{maximized}(x))$

Table 5: Desktop management actions

Desktop-level application management

In the final domain we consider a set of DCOP services common to a wide range of applications, that manipulate desktop applications as "entities," rather than providing interfaces to application-specific functions. As we noted above, DCOP services require an executing instance of the application. In particular, we illustrate plan-level control over application execution, and manipulate a set of properties common to most windows-based application interfaces.

Table 5 shows five desktop services. The first service, *find*(x), is not strictly a DCOP service but abstracts behaviour provided by the DCOP mechanism itself: DCOP can determine whether a particular KDE application is running or not. Here, our PKS encoding is a wrapper around this service. The second service, *kdeinit_exec*(x) is provided by an application called *klauncher* which is able to control application execution. In this case, the service directs an instance of a particular KDE application x to be started. The remaining three services, *minimize*(x), *maximize*(x), and *restore*(x), fall into a class of common services provided by many GUI-based applications, and are similar to the widget controls found on applications with windowed interfaces.[6]

The PKS encoding of *find* in Table 5 models its effects as a sensing action that adds knowledge of *running* to the K_w database. Thus, after adding *find*(x) to a plan PKS can construct a conditional branch based on *running* to reason about the execution state of individual applications.

For instance, consider a scenario described by the initial database $K_f = \{\textit{KdeApp}(\textit{app1}), \textit{KdeApp}(\textit{app2}), \textit{KdeApp}(\textit{app3}), \textit{running}(\textit{app2})\}$, where all other databases are empty. Thus, PKS knows about three applications, *app1*, *app2*, and *app3*, and knows that *app2* is already running. If we present PKS with the goal $K(\neg\textit{minimized}(\textit{app2})) \wedge K(\textit{maximized}(\textit{app3})) \wedge \forall^K x.K(\textit{KdeApp}(x)) \Rightarrow K(\textit{running}(x))$, i.e., ensure all three applications are running, *app2* is minimized, and *app3* is maximized, then one possible plan PKS generates is:

[6] Although we group these services together, each application typically provides its own set of DCOP *mainwindow* services.

external::dcop::find(app1)
external::dcop::find(app3)
apps::mainwindow::restore(app2)
branch(running(app3))
K^+:
    apps::mainwindow::maximize(app3)
    branch(running(app1))
    K^+:
        nil
    K^-:
        klauncher::kdeinit_exec(app1)
K^-:
    klauncher::kdeinit_exec(app3)
    apps::mainwindow::maximize(app3)
    branch(running(app1))
    K^+:
        nil
    K^-:
        klauncher::kdeinit_exec(app1)

The plan first determines whether or not *app1* and *app3* are running by using *find*, which adds *running*(*app1*) and *running*(*app3*) to K_w. Since *app2* is already running, *restore*(*app2*) ensures that it is not minimized. At the first branch point, PKS reasons about the two possibilities for *running*(*app1*). Along the K^+ branch, *running*(*app3*) is true and so it can simply maximize the application; along the K^- branch, $\neg$*running*(*app3*) is true and so *app3* must first be started using *kdeinit_exec*(*app3*) before it can be maximized. The remaining subplan along each branch is then the same: PKS must again reason about the state of *running*(*app1*) by introducing a new branch. If *app1* is running then nothing needs to be done. Otherwise, the application must be started using *kdeinit_exec*(*app1*).

Execution of DCOP-based plans

All of the examples we describe in the previous section are quickly generated by PKS, typically in less than 1 second. What we are left with, however, is a plan that describes DCOP-level actions with plan-level control directives like *branch*, K^+, K^-, etc. Thus, such plans must first be converted into an appropriate form before they can be executed.

For sequential plans with fully instantiated arguments (i.e., no functions), such as those in our first example domain, the job of transforming such plans into an executable form is straightforward: we can syntactically transform each action *app:category::service(arguments)* into the form:

```
dcop app category service arguments
```

and run the resulting statements as a simple shell script. For more complex plans, such as those with branches, we use the Perl interface to DCOP. By doing so we can use if-else control structures in place of plan branches, and standard variables to denote plan-level run-time variables, which can be assigned the execution-time return values of DCOP calls.

As part of our postprocessing stage, we also ensure that the appropriate application instances have been started for the services we require in a plan. While we have investigated automatically starting applications at the planning level, in the style of our last example domain, we have also done so by simply scanning a generated plan, extracting the list of applications used, and adding a prefix to the plan with the appropriate *klauncher::kdeinit_exec* actions.

Furthermore, we do not currently perform any plan execution monitoring, but instead we simply verify the final outcome of executed plans as succeeding or failing. We leave the plan monitoring task for future work.

Discussion

One of the difficulties arising from using a language like DCOP as the basis for an action representation is that DCOP was designed primarily for programming and script writing. As such, its semantics are targeted at the application programmer and do not provide the means necessary for distinguishing between application-specific operations or indicating the relationships between the information such services provide.[7] Although some services are commonly available across many applications (e.g., the *mainwindow* services), developers are free to include whatever services they deem necessary, making it difficult to automate the process of encoding actions from DCOP services.

Recent work has addressed some of these deficiencies for the desktop. For instance, the NEPOMUK project[8] aims to develop a "Social Semantic Desktop" (see, e.g., (Richter, Völkel, & Haller 2005; Sauermann *et al.* 2006)) that seeks to enhance the standard desktop model by attaching meta-level meaning to desktop information and services, making it easier to exchange information between other desktops and users—and more manageable by automated means. In particular, an interesting NEPOMUK subproject aims to adapt these ideas to the KDE desktop.[9]

DCOP itself goes a long way towards overcoming some of the practical issues concerning software interoperability, by providing a common language for interacting with applications. An interesting observation is the realization that the interface to web services is typically through the desktop. This observation is particularly important when we consider that many applications do not distinguish between "desktop" and "network." Instead, desktop applications routinely use network transparency to shelter users from the trouble of differentiating between a wide range of accessible files, services, and devices. (For instance, many media players seamlessly play both local files and remote streams, and web browsers like Firefox or Konqueror also double as local file managers.) Using desktop services to access web services means that we can often ignore issues related to the transport medium itself (i.e., the network), since such services are viewed by the desktop interface as "local."

DCOP is also important due to the large number of KDE

[7] It is often easy to determine how services should be modelled, but not how the information provided by these services relates to other services. For instance, services that return Boolean values can usually be modelled using K_w, while those that return strings can be modelled using functions and K_v.

[8] The project website can be found at `nepomuk.semanticdesktop.org`.

[9] See `nepomuk-kde.semanticdesktop.org`.

applications that already use this interface. (Some of these applications provide no software interface other than DCOP and the standard graphical interface.) Moreover, a successor to DCOP called D-BUS, which is based on DCOP, is being proposed as a desktop-independent standard by the open source community (Pennington, Carlsson, & Larsson 2006).

There is also a question as to how planning technology should be incorporated into KDE. An interesting approach is the prospect of developing a KDE component that provides planning services (possibly through DCOP) to users and other applications. How goals are conveyed to such a component, and in what form, remains an open problem.

Although we present a set of examples which demonstrate the flavour of the plans we can already generate (and some interesting behaviour), work still needs to be done to extend our examples to model more challenging aspects of desktop domains, in order to determine the scalability of our approach. Some domains require additional forms of knowledge. For instance, searching a media player's song database for all the tracks by a particular artist produces an instance of local closed world knowledge, which could be modelled by PKS's LCW database. More work is also needed to enhance PKS's use of functions and K_v knowledge, which together with arithmetic operations are often required for resource management in the domains we have considered. We have also begun experimenting with interleaving planning and execution. Although PKS manages many types of information effectively, we have found that it is often useful to execute partial plans in order to fill in PKS's databases before constructing large plans, especially in response to large domains. Based on our preliminary results, however, we remain positive that the knowledge-level approach to planning can be successful in operating system environments.

Conclusions

In this paper we investigated the use of knowledge-level planning techniques in a desktop services domain. Since the desktop interface is the natural interface to many applications, and such applications are often suited to tasks for which there might not be existing alternatives, the challenge of harnessing these services for automated planning remains a worthwhile task. While we have focused on a particular inter-application communication language (DCOP) in a particular desktop environment (KDE), we have also focused on meeting the planning needs of real services provided by existing applications. As such, we believe that this domain is a useful testbed for furthering our goal of constructing agents that can operate in complex operating system environments.

Acknowledgements

The work reported in this paper was partially funded by the European Commission through the PACO-PLUS project (FP6-2004-IST-4-27657).

References

Brown, P.; Ettrich, M.; Meine, H.; and Jansen, T. 2003. *DCOP: Desktop COmmunications Protocol*. http://developer.kde.org/document-ation/other/dcop.html.

Bryant, R. E. 1992. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys* 24(3):293–318.

Etzioni, O., and Weld, D. 1994. A softbot-based interface to the internet. *Communications of the ACM* 37(7):72–76.

Etzioni, O.; Hanks, S.; Weld, D.; Draper, D.; Lesh, N.; and Williamson, M. 1992. An approach to planning with incomplete information. In *Proc. of KR-92*, 115–125. Cambridge, MA: Morgan Kaufmann Publishers.

Etzioni, O.; Levy, H. M.; Segal, R. B.; and Thekkath, C. A. 1993. OS agents: Using AI techniques in the operating system environment. Technical Report UW-CSE-93-04-04, University of Washington.

Etzioni, O.; Golden, K.; and Weld, D. 1994. Tractable closed world reasoning with updates. In *Proc. of KR-94*, 178–189. Bonn, Germany: Morgan Kaufmann Publishers.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Martínez, E., and Lespérance, Y. 2004. Web service composition as a planning task: Experiments using knowledge-based planning. In *Proceedings of the ICAPS-04 Workshop on Planning and Scheduling for Web and Grid Services*.

McIlraith, S., and Son, T. C. 2002. Adapting Golog for composition of semantic web services. In *Proc. of KR-2002*, 482–493. Morgan Kaufmann Publishers.

Pednault, E. P. D. 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. of KR-89*, 324–332. Morgan Kaufmann Publishers.

Pennington, H.; Carlsson, A.; and Larsson, A. 2006. *D-Bus Specification, Version 0.12*. http://dbus.freedesktop.org/doc/dbus-specification.html.

Petrick, R. P. A., and Bacchus, F. 2002. A knowledge-based approach to planning with incomplete information and sensing. In *Proc. of AIPS-2002*, 212–221. AAAI Press.

Petrick, R. P. A., and Bacchus, F. 2004. Extending the knowledge-based approach to planning with incomplete information and sensing. In *Proc. of ICAPS-04*, 2–11. AAAI Press.

Pistore, M.; Marconi, A.; Bertoli, P.; and Traverso, P. 2005. Automated composition of web services by planning at the knowledge level. In *Proc. of IJCAI-05*, 1252–1259.

Richter, J.; Völkel, M.; and Haller, H. 2005. DeepaMehta – a semantic desktop. In Decker, S.; Park, J.; Quan, D.; and Sauermann, L., eds., *1st Workshop on The Semantic Desktop. 4th International Semantic Web Conference*.

Sauermann, L.; Grimnes, G. A.; Kiesel, M.; Fluit, C.; Maus, H.; Heim, D.; Nadeem, D.; Horak, B.; and Dengel, A. 2006. Semantic desktop 2.0: The gnowsis experience. In *Proc. of the ISWC Conference*.

Weld, D. S.; Anderson, C. R.; and Smith, D. E. 1998. Extending Graphplan to handle uncertainty & sensing actions. In *Proc. of AAAI-98*, 897–904. AAAI Press.

Wheeler, S. 2003. KDE scripting with DCOP: Boost your efficiency. *Linux Magazine* 36:46–48.