Planning for Desktop Services

Ron Petrick

School of Informatics
University of Edinburgh
Edinburgh, Scotland, UK

Presented at the ICAPS'07 Workshop on Moving Planning and Scheduling Systems into the Real World, Providence, Rhode Island, USA (23 September 2007)
Motivation

- A user’s experience in an **operating system environment** is typically reflected in the use of **desktop applications**
  - E-mail clients
  - Web browsers
  - Media players
  - Document viewers
  - ...
- Desktop applications provide an abstracted interface to the **services** the OS can perform through the application
  - Play a music file
  - Display a document
  - ...
- An agent (human or artificial) should be able to use such services to achieve its goals or objectives
- Planning?
Planning in operating system environments

- Not a new idea
- Softbot project (Etzioni et al. 1993, Etzioni & Weld 1994)
  - UNIX commands as planning operators
  - Effectors: `ftp`, `lpr`, …
  - Sensors: `ls`, `finger`, …
- Additional UNIX domains
  - (Petrick & Bacchus 2002)
  - …
- Planning for web services, web service composition
  - (McIlraith & Son 2002)
  - (Martínez & Lespérance 2004)
  - (Pistore et al. 2005)
  - …
How do desktop services differ?

• UNIX commands vs. desktop services
  – Many desktop applications tend to operate at a much higher level of abstraction
  – E.g., *ls* (files/directories) vs. media player (artist/title/genre)
  – Command line interaction vs. standard windowed interface

• Web services vs. desktop services
  – Operation of desktop applications tends to be “local”
  – Some desktop applications provide seamless interfaces to external services
  – Access to the web is usually through the desktop
  – E.g., web browsers, media players that play remote streams

• Desktop services often occupy a “middle ground”
Our approach

• Motivate desktop services as a challenging (and interesting!) real-world domain for planning
  – Planning with incomplete information and sensing
  – Planning with resources
  – Planning with …?

• Apply knowledge-level techniques to these problems
  – “Planning with Knowledge and Sensing”
Our approach...

- Focus on real applications running in a real desktop environment
  - *K Desktop Environment* (KDE)

- Use existing facilities as much as possible for communicating with applications
  - *Desktop COmmunication Protocol* (DCOP)
- Open source, freely available desktop for UNIX-like systems
- Standard applications, OSS community-driven projects
A novel feature of KDE is its inter-application communication language **DCOP**.

DCOP allows a KDE application to publish its **services** for interoperability with other running programs.

DCOP services are often similar to functions a user can perform through the standard (graphical) interface.

Underlying support for DCOP is provided by KDE itself (cf. CORBA, ICE, …).

Multiple DCOP interfaces:
- C++, C, Perl, Python, …
- KDE application `kdcop`
- Command line interface (scripts)
Example DCOP interface

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

$ dcop amarok player play
Planning with DCOP actions

• DCOP services typically include functions for changing and querying an application’s state
• Leverage “abstracted” services for rich planning domains
  – Use planning operators that closely correspond to DCOP services
  – Generate plans that link services provided by real desktop applications
• Similar to DCOP scripts but with plan-level control directives
• A variety of target languages for plan execution
• Preliminary examples: planning with incomplete information and sensing using PKS
A “knowledge-level” conditional planner that builds plans based on knowledge state (Petrick & Bacchus 2002, 2004)

Based on an extension of the STRIPS representation

Planner’s knowledge state is represented by 5 databases, each of which models a different type of knowledge

Knowledge is restricted for tractable reasoning

Non-propositional features like functions and run-time variables
PKS databases

• \( K_f \): knowledge of positive and negative facts (no CWA)
  \[ p \in K_f : \text{the agent knows } p \]

• \( K_w \): knowledge of binary sensing effects
  \[ \phi \in K_w : \text{the agent knows } \phi \text{ or knows } \neg \phi \]

• \( K_v \): knowledge of function values, multi-valued sensing effects
  \[ f \in K_v : \text{the agent knows the value of } f \]

• \( K_x \): exclusive-or knowledge
  \[ (\ell_1|\ell_2|\ldots|\ell_n) \in K_x : \text{exactly one of the } \ell_i \text{ must be true} \]

• \( LCW \): local closed world information (Etzioni et al. 1994)
PKS actions

<table>
<thead>
<tr>
<th>Action</th>
<th>Preconditions</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pickup(x)</td>
<td>$K(handempty)$</td>
<td>$add(K_f, holding(x))$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$add(K_f,\neg handempty)$</td>
</tr>
<tr>
<td>inspect(x)</td>
<td>$K(holding(x))$</td>
<td>$add(K_w, fragile(x))$</td>
</tr>
</tbody>
</table>

- Preconditions are represented as databases queries
- Effects update the knowledge state (databases), rather than the world state, in a STRIPS-like manner
- Easy to compute new knowledge states by forward chaining
- $K_v$ and $K_w$ model information returned from sensing that can be used to introduce conditional branch points into a plan
Example 1: Amarok media player
### Actions for controlling a media player

<table>
<thead>
<tr>
<th>Action</th>
<th>Precond.</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>amarok::playlist::</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addMedia(x)</td>
<td>( K(\text{media}(x)) )</td>
<td>( \text{add}(K_f, \text{inplaylist}(x)) ) ( \text{add}(K_f, \text{track}(x) = \text{total} + 1) ) ( \text{add}(K_f, \text{total} = \text{total} + 1) )</td>
</tr>
<tr>
<td>clearPlaylist</td>
<td></td>
<td>( \text{add}(K_f, \text{total} = 0) ) ( \text{add}(K_f, \text{current} = 0) ) ( \forall K_x. \text{del}(K_f, \text{inplaylist}(x)) ) ( \forall K_x. \text{del}(K_f, \text{track}(x)) )</td>
</tr>
<tr>
<td><strong>amarok::player::</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>play</td>
<td></td>
<td>( K(\text{total} &gt; 0) \Rightarrow \text{add}(K_f, \text{playing}) ) ( K(\text{current} = 0) \Rightarrow \text{add}(K_f, \text{current} = 1) )</td>
</tr>
<tr>
<td>next</td>
<td></td>
<td>( K(\text{total} &gt; \text{current}) \Rightarrow \text{add}(K_f, \text{current} += 1) )</td>
</tr>
</tbody>
</table>
Playing a media file

Initial state: \( K_f = \{media(track1), media(track2), media(track3)\} \)

Goal: \( K(\text{playing}) \)

Plan

- `amarok::playlist::clearPlaylist`
- `amarok::playlist::addMedia(track1)`
- `amarok::player::play`
Playing the last file in a playlist

Initial state: $K_f = \{\text{media(track1)}, \text{media(track2)}, \text{media(track3)}\}$

Goal: $K(\text{playing}) \land K(\text{total} = \text{current}) \land \forall^K x. K(\text{media}(x)) \Rightarrow K(\text{inplaylist}(x))$

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`
Example 2: Amarok + Knotify
Actions for querying the state of a media player

<table>
<thead>
<tr>
<th>Action</th>
<th>Precond.</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>amarok::player::</td>
<td></td>
<td>add($K_w$, result(playing))</td>
</tr>
<tr>
<td>isPlaying</td>
<td></td>
<td></td>
</tr>
<tr>
<td>isOsdEnabled</td>
<td></td>
<td>add($K_w$, result(osd))</td>
</tr>
<tr>
<td>knotify::</td>
<td></td>
<td></td>
</tr>
<tr>
<td>notify($x, y$)</td>
<td>$K(service(x))$</td>
<td>add($K_f$, notified(?x))</td>
</tr>
<tr>
<td></td>
<td>$K(msgtosend(x, y))$</td>
<td></td>
</tr>
</tbody>
</table>

Domain specific update rules

- $K(result(x)) \Rightarrow add(K_f, msgtosend(x, true))$
- $K(\neg result(x)) \Rightarrow add(K_f, msgtosend(x, false))$
Notifying a user of a media player’s state

Initial state: $K_f = \{service(playing), service(osd)\}$

Goal: $\forall^K x. K(service(x)) \Rightarrow K(notified(x))$

Plan

```
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))
  $K^+ : knotify::notify(playing, true)$
  $K^- : knotify::notify(playing, false)$
```

Ron Petrick  Planning for Desktop Services / Presented 2007-09-23
Example 3: Kweather applet

Weather Report - Edinburgh Airport - United Kingdom

Latest data from 23 September 2007 02:20

- Few clouds at 914 metres
- Scattered clouds at 1371 metres

Temperature: 15°C  Dew Point: 11°C
Air Pressure: 1007 hPa  Rel. Humidity: 77%
Wind Speed: 31 km/h SW
Sunrise: 07:00  Sunset: 19:11
Actions providing a desktop interface to a web service

<table>
<thead>
<tr>
<th>Action</th>
<th>Precond.</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>KWeatherService::WeatherService::</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>stationCode(x)</code></td>
<td>$K_v(x)$</td>
<td>$add(K_w, validCode)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$add(K_v, code)$</td>
</tr>
<tr>
<td><code>temperature(x)</code></td>
<td>$K(validCode)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$K(code = x)$</td>
<td>$add(K_v, stationTemp)$</td>
</tr>
<tr>
<td><code>external::kdialog::</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>inputbox</code></td>
<td></td>
<td>$add(K_v, stationName)$</td>
</tr>
</tbody>
</table>
Querying the temperature at a weather station

Initial state: all databases empty

Goal: $K_v(stationTemp) \lor K(\neg validCode)$

Plan

```
external::kdialog::inputbox
KWeatherService::WeatherService::stationCode(stationName)
branch(validCode)
  $K^+:$
    KWeatherService::WeatherService::temperature(stationCode)
  $K^-:$
    nil
```
Other examples of DCOP actions

<table>
<thead>
<tr>
<th>Action</th>
<th>Precond.</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>external::dcop::</td>
<td></td>
<td></td>
</tr>
<tr>
<td>find(x)</td>
<td>$K(KdeApp(x))$</td>
<td>add($K_w$, running(x))</td>
</tr>
<tr>
<td>klauncher::</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kdeinit_exec(x)</td>
<td>$K(KdeApp(x))$</td>
<td>add($K_f$, running(x))</td>
</tr>
<tr>
<td></td>
<td>$K(\neg \text{running}(x))$</td>
<td></td>
</tr>
<tr>
<td>app::mainwindow::</td>
<td></td>
<td></td>
</tr>
<tr>
<td>minimize(x)</td>
<td>$K(\text{running}(x))$</td>
<td>add($K_f$, minimized(x))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>add($K_f$, $\neg \text{maximized}(x)$)</td>
</tr>
<tr>
<td>maximize(x)</td>
<td>$K(\text{running}(x))$</td>
<td>add($K_f$, maximized(x))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>add($K_f$, $\neg \text{minimized}(x)$)</td>
</tr>
<tr>
<td>restore(x)</td>
<td>$K(\text{running}(x))$</td>
<td>add($K_f$, $\neg \text{minimized}(x)$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>add($K_f$, $\neg \text{maximized}(x)$)</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Execution of DCOP-based plans

• Plans include DCOP actions with plan-level control directives
• Fully instantiated sequential plans without functions
  – Syntactically transform each action into the form
    
dcop app category service arguments
  – Run plan as a simple shell script
• Plans with branches and functions
  – Transform branches into if-else structures
  – Use standard variables in place of run-time variables
  – Execute plan as a program or structured script
• Ensure application start up (currently no plan monitoring)
• More work needed!

Ron Petrick  Planning for Desktop Services / Presented 2007-09-23
Observations and open questions

• DCOP is primarily a programming/scripting language
  - Semantics are targeted at the application developer
  - Automatically compile DCOP functions to planning operators?
  - NEPOMUK “Social Semantic Desktop” + KDE (Richter et al. 2005, Sauermann et al. 2006)

• DCOP as a shared language for application interaction
  - Many applications do not distinguish “desktop” from “network”
  - Desktop is often the interface to web services

• A successor to DCOP called D-BUS is being proposed as a desktop-independent standard by the open source community

• How to incorporate a planner into a desktop environment?
• How should goals be conveyed to a planner?
Conclusions and future work

• Focus on existing interfaces to real applications in a real desktop environment
• Desktop interface provides a natural and abstract interface to many applications
• Planning results are preliminary; more work needed to extend examples and determine scalability of approach using PKS
• Many examples require more complex knowledge: $K_x$ (exclusive-OR), $LCW$ (local closed world), functions, …
• Desktop environment is a challenging (and interesting!) testbed for planning with incomplete information, sensing and information gathering actions, resources, …
Web links

- K Desktop Environment (KDE)
  http://www.kde.org/

- Desktop COmmunications Protocol (DCOP)
  http://developer.kde.org/documentation/other/dcop.html

- D-Bus specification
  http://dbus.freedesktop.org/doc/dbus-specification.html

- Amarok music player
  http://amarok.kde.org/

- PKS
  http://homepages.inf.ed.ac.uk/rpetrick/research/pks/


