Models of the System

Overview

Standard Formalisms
software engineering notations used to specify the required behaviour of specific interactive systems

Interaction Models
special purpose mathematical models of interactive systems, used to describe usability properties at a generic level

Status/Event Analysis
an example of an engineering level method drawing on both formal modelling and naïve psychology

Relationship with dialogue

Dialogue modelling is linked to semantics.

System semantics affects the dialogue structure.

But the bias is different.

Rather than dictate what actions are legal, these formalisms tell what each action does to the system.

Irony

Computers are inherently mathematical machines.

Humans are not.

Formal techniques are well accepted for cognitive models of the user and the dialogue (what the user *should do*).

Formal techniques are not yet well accepted for dictating what the system should do *for the user*!

General computational formalisms

Standard software engineering formalisms can be used to specify an interactive system.

Referred to as *formal methods*

Model based describe system states and operations

- Z, VDM

Algebraic describe effects of sequences of actions

- OBJ, Larch, ACT-ONE

Extended logics describe when things happen and who is responsible

- temporal and deontic logics

The uses of SE formal notations

For communication

- common language
- remove ambiguity (possibly)
- succinct and precise

For analysis

- internal consistency
- external consistency
  - with eventual program
  - with respect to requirements (safety, security, HCI)
- specific versus generic

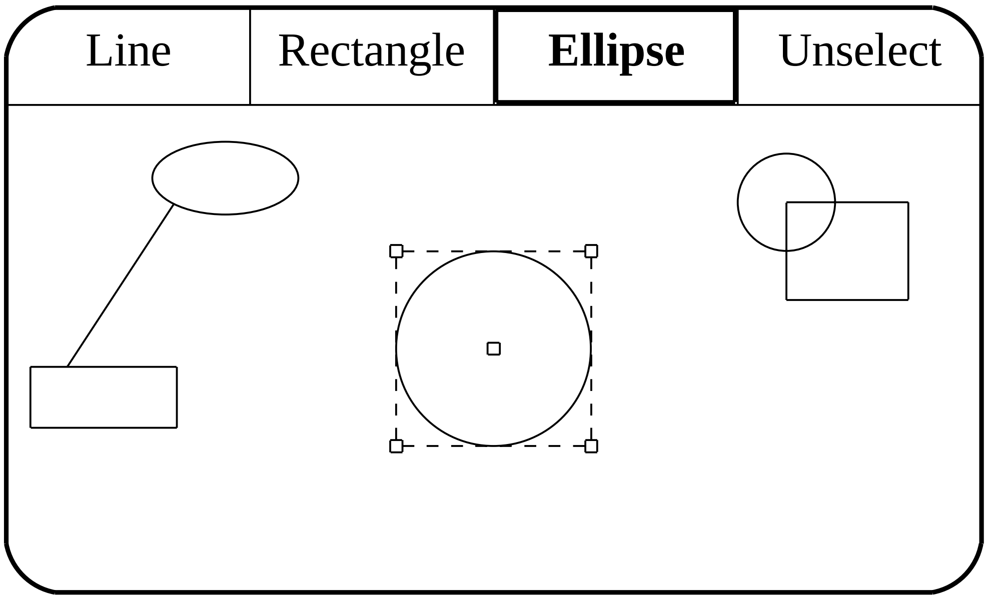
Model based formalisms

Mathematical counterparts to common programming constructs

Programming	Mathematics
types	sets
basic types	basic sets
constructed types	constructed sets
records	unordered tuples
lists	sequences
functions	functions
procedures	relations

The model based method

Example: a graphic drawing package



Points are ordered pairs.

$$Point == \mathbb{N} \times \mathbb{N}$$

Shapes can be of varying types.

$$Shape\_type == line | ellipse | rectangle$$

More type definitions

A graphic object is defined by its shape type, width, height, and centre position.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Shape}} \\ \hline
type : Shape\_type \\
width, height : \mathbb{N} \\
centre : Pt \\ \hline
\end{array}$$

A collection of graphic objects can be identified by a ‘lookup dictionary’

$$[Id]$$
$$Shape\_dict == Id \nrightarrow Shape$$

Defining the state

The system state contains a dictionary of created objects and a set of selected objects.

$$\begin{array}{|l}
\hline
\textit{State} \\
\textit{shapes} : \textit{Shape\_dict} \\
\textit{selection} : \mathbb{P}\,\textit{Id} \\
\hline
\textit{selection} \subseteq \operatorname{dom} \textit{shapes} \\
\hline
\end{array}$$

Initially, there are no shapes in the dictionary.

$$\begin{array}{|l}
\hline
\textit{Init\_State} \\
\textit{State}' \\
\hline
\textit{shapes}' = \{\} \\
\hline
\end{array}$$

Defining operations

State change is represented as two copies of the state

before — $State$
after — $State'$

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\Delta State\qquad\qquad\qquad\qquad}} \\
State \\
State' \\
\hline
\end{array}$$

The $Unselect$ operation deselects any selected objects.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{Unselect\qquad\qquad\qquad\qquad}} \\
\Delta State \\
\hline
selection' = \{\} \\
shapes' = shapes \\
\hline
\end{array}$$

Interface issues

Framing problem "everything else stays the same" can be complicated with state invariants

Internal consistency do operations define any legal transition?

External consistency must be formulated as theorems to prove

Clear for refinement, not so for requirements

Separation of system functionality and presentation is not explicit

Algebraic notations

Model based notations emphasise constructing an explicit representations of the system state.

Algebraic notations provide only implicit information about the system state.

Model based operations are defined in terms of their effect on system components.

Algebraic operations are defined in terms of their relationship with the other operations.

Return to graphics example

types

State, Pt

operations

$init : \rightarrow State$

$make\_ellipse : Pt \times State \rightarrow State$

$move : Pt \times State \rightarrow State$

$unselect : State \rightarrow State$

$delete : State \rightarrow State$

axioms

for all $st \in State;\ p \in Pt\ \bullet$

1. $delete(make\_ellipse(st)) = unselect(st)$
2. $unselect(unselect(st)) = unselect(st)$
3. $move(p, unselect(st)) = unselect(st)$

Issues for algebraic notations

Ease of use a different way of thinking than traditional programming

Internal consistency are there any axioms which contradict others?

External consistency with respect to executable system less clear

External consistency with respect to requirements is made explicit and automation possible

Completeness is every operation completely defined?

Extended logics

Model based and algebraic notations make extended use of propositional and predicate logic.

Propositions expressions made up of atomic terms $p, q, r, \ldots$ composed with $(,), \wedge, \vee, \neg\ , \Rightarrow$, etc.

Predicates propositions with variables, e.g., $p(x)$ and quantified expressions $\forall, \exists$.

These are not convenient for expressing time, responsibility and freedom, notions sometimes needed for HCI requirements.

Temporal logics

Time considered as succession of events

Basic operators:

always	$\Box$	$\Box$(G funnier than A)
eventually	$\Diamond$	$\Diamond$(G understands A)
never	$\Box\neg$	$\Box\neg$ (rains in So. Cal.)

Other bounded operators:

p *until* q	weaker than $\Box$
p *before* q	stronger than $\Diamond$

Explicit time

These temporal logics do not explicitly mention time, so some requirements cannot be expressed.

Active research area, but not so much with HCI

Gradual degradation more important than time-criticality

Myth of the infinitely fast machine

Deontic logics

For expressing responsibility, obligation between separate agents (e.g., the human, the organisation, the computer)

permission	*per*
obligation	*obl*

For example,

$$\begin{array}{l} \mathit{owns}(\text{ Jane, file `fred' }) \Rightarrow \\ \quad \mathit{per}(\text{ Jane, } \mathit{request}(\text{`print fred'})) \end{array}$$

$$\begin{array}{l} \mathit{performs}(\text{ Jane, } \mathit{request}(\text{`print fred'})) \Rightarrow \\ \quad \mathit{obl}(\text{ lp3, } \mathit{print}(\text{file `fred'})) \end{array}$$

Issues for extended logics

Safety properties stipulating that bad things do not happen

Liveness properties stipulating that good things do happen

Executability versus expressiveness easy to specify impossible situations; difficult to express executable requirements; settle for eventual executable

Group issues and deontics obligations for single-user systems have personal impact; for groupware, we must consider implications for other users.

Interaction models

General computational models were not designed with the user in mind.

We need models that sit between the software engineering formalism and our understanding of HCI.

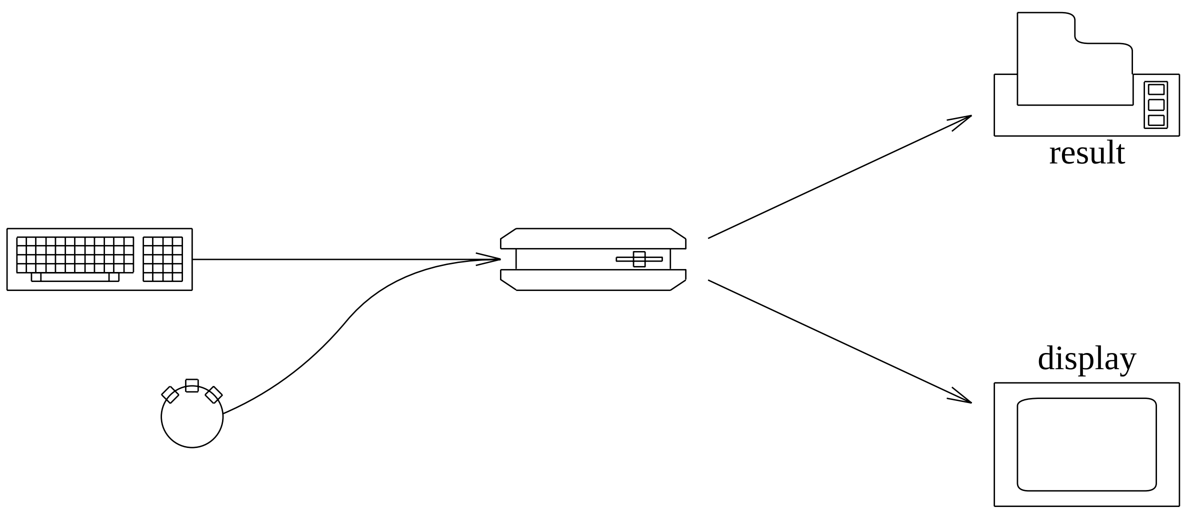
formal the PIE model for expressing general interactive properties to support usability

informal interactive architectures (MVC, PAC, ALV) to motivate separation and modularisation of functionality and presentation

semi-formal status-event analysis for viewing a slice of an interactive system that spans several layers

The PIE model

A black-box model



$$\begin{tikzcd}
 & & R \\
P \arrow[r, "I"] & E \arrow[ur, "result"] \arrow[dr, "display"'] & \\
 & & D
\end{tikzcd}$$

More formally

$$[C, E, D, R]$$
$$P == \operatorname{seq} C$$

$$\begin{array}{|l}
I : P \rightarrow E \\
display : E \rightarrow D \\
result : E \rightarrow R
\end{array}$$

Alternatively, we can derive a state transition function from the PIE.

$$\begin{array}{|l}
doit : E \times P \rightarrow E \\
\hline
doit(I(p), q) = I(p \frown q) \\
doit(doit(e, p), q) = doit(e, p \frown q)
\end{array}$$

Expressing properties

WYSIWYG

What does this really mean, and how can we test product X to see if it satisfies a claim that it is WYSIWYG?

Limited scope general properties which support WYSIWYG.

Observability what you can tell about the current state of the system from the display

Predictability what you can tell about the future behaviour

Observability and predictability

Two possible interpretations of WYSIWYG:

What you see is what you:

- *will get* at the printer
- *have got* in the system

Predictability is a special case of observability

Formally

$$\begin{tikzcd}
 & & R \\
P \arrow[r, "I"] & E \arrow[ur, "result"] \arrow[dr, "display"'] & \\
 & & D
\end{tikzcd}$$

Determining result from display:

$$\begin{array}{l} \exists\, f : D \rightarrow R \bullet \\ \quad \forall\, e : E \bullet f(display(e)) = result(e) \end{array}$$

Determining effect from display:

$$\begin{array}{l} \exists\, f : D \rightarrow E \bullet \\ \quad \forall\, e \in E \bullet f(display(e)) = e \end{array}$$

Relaxing the property

$$\begin{tikzcd}
 & & R \\
P \arrow[r, "I"] & E \arrow[ur, "result"] \arrow[dr, "observe"'] & \\
 & & O \arrow[r] & D
\end{tikzcd}$$

$$\begin{array}{l} \exists\, f : O \rightarrow R \bullet \\ \quad \forall\, e \in E \bullet f(observe(e)) = result(e) \end{array}$$

$$\begin{array}{l} \exists\, f : O \rightarrow E \bullet \\ \quad \forall\, e \in E \bullet f(observe(e)) = e \end{array}$$

Reachability and undo

Reachability — getting from one state to another.

$$\forall\, e, e' \in E \bullet \exists\, p \in P \bullet doit(e, p) = e'$$

Too weak

Undo — reachability applied between current state and last state.

$$\forall\, c \in C \bullet doit(e, c \frown undo) = e$$

Impossible except for very simple system with at most two states!

Better models of *undo* treat it as a special command to avoid this problem

Issues for PIE properties

Insufficient define necessary but not sufficient properties for usability.

Generic can be applied to any system

Proof obligations for system defined in SE formalism

Scale how to prove many properties of a large system

Scope limiting applicability of certain properties

Insight gained from abstraction is reusable

Status/event analysis

semi-formal technique

“engineering” level analysis

based on formal models

uses naïve psychology

clocks and calendars as example

status – analogue watch face

event – an alarm

Properties of events

status change event

- the passing of a time

actual and perceived events

- usually some gap

polling

- glance at watch face
- status change becomes perceived event

granularity

- birthday – days
- appointment – minutes

Design implications

actual/perceived lag. . .
matches application timescale?

too slow

- response to event too late
- e.g., power plant emergency

too fast

- interrupt more immediate task
- e.g., stock level low

Naïve psychology

Predict where the user is looking

mouse – when positioning

insertion point – intermittently when typing

screen – if you're lucky

Immediate events

audible bell – when in room (and hearing)

peripheral vision – movement or large change

Closure

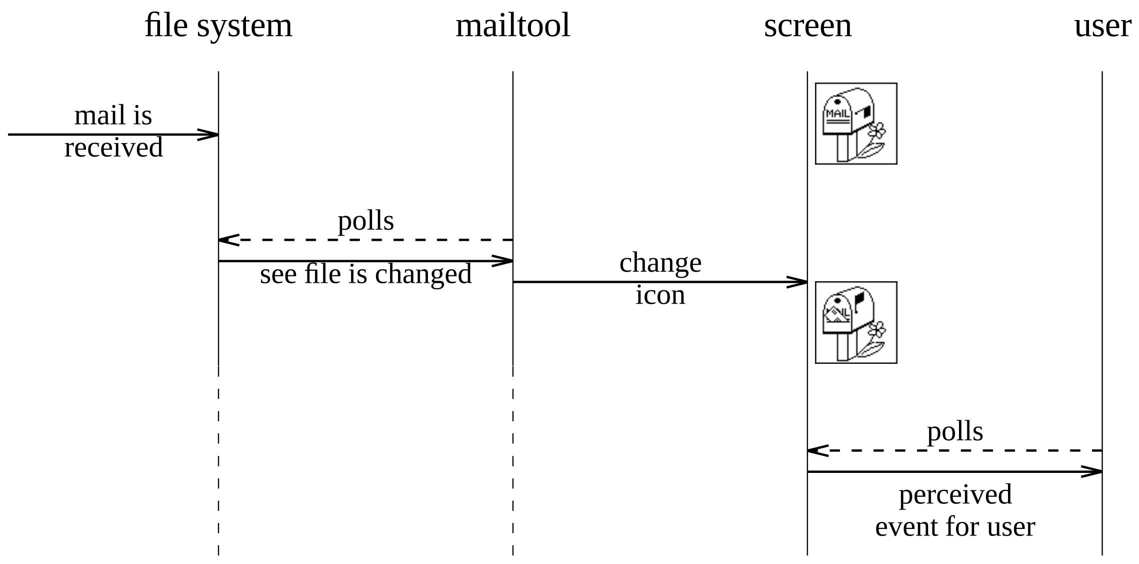
lose attention (inc. mouse)

concurrent activity

Example – email interface

mail has arrived!

timeline at each level



Perceived event in minutes – not guaranteed

alternative		**timescale**
explicit examination	–	hours/days
audible bell	–	seconds

want minutes – guaranteed

Example – screen button widget (i)

screen button often missed, . . .
but, error not noticed

a common widget, a common error: Why?

Closure

mistake likely – concurrent action

not noticed – semantic feedback missed

Solution

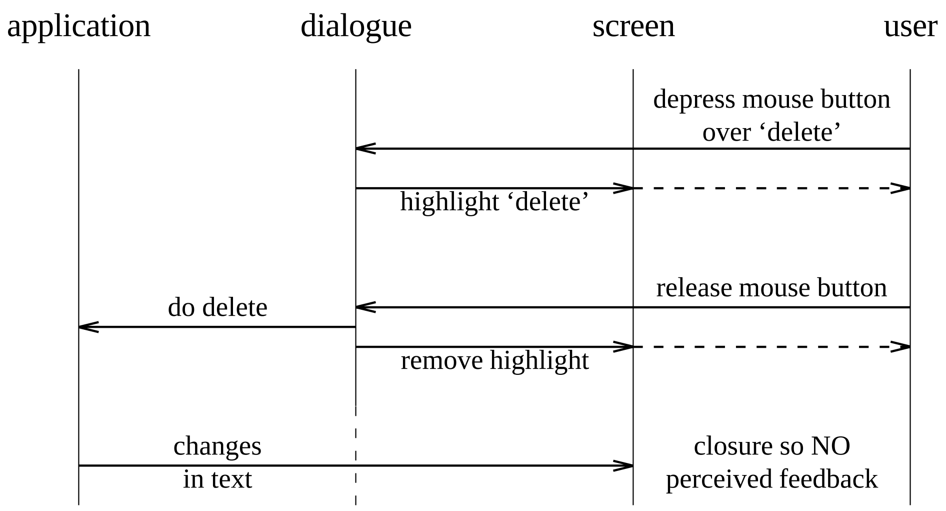
widget feedback for application event

a *perceived event* for the user

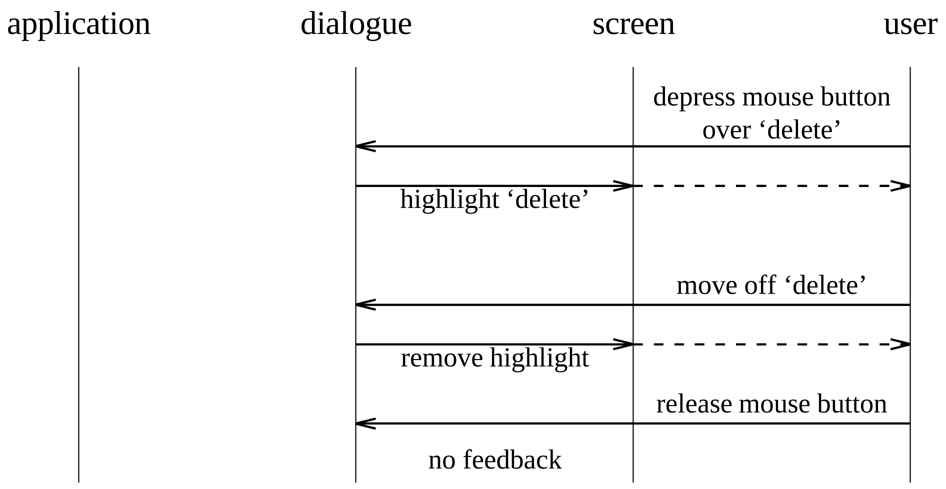
N.B., an expert slip – testing doesn't help

Screen button widget (ii)

a HIT



or a MISS


application
dialogue
screen
user
depress mouse button over 'delete'
highlight 'delete'
move off 'delete'
remove highlight
release mouse button
no feedback