App Development with Talkamatic Dialogue Manager

June 23, 2015

Alex Berman <alex@talkamatic.se>

Outline

- Introduction to TDM
- Technical architecture
- App development philosophy
- App formalism
- Test-driven app development

Introduction to TDM

- TDM = Talkamatic Dialogue Manager
- Enables natural, easy-to-use dialogue
- Features low distraction and high efficiency
- Allows rapid prototyping thanks to built-in dialogue design

Introduction to TDM

- TDM is generic
- Easy to add new apps / domains
- Built-in multimodality (GUI + speech)
- Supports English and Swedish
  - Soon Italian, French, Dutch, German and Spanish
- Supports many different speech recognizers and text-to-speech engines

Technical architecture

- TDM runtime consists of a backend and frontend
- Backend supports multiple simultaneous frontends
- Backend can run in the cloud or locally
  - Cross-platform (Python)
- Supported frontends:
  - Android
  - IOS (beta)
  - Text I/O via console (cross-platform, for development)

Technical architecture

- TDM SDK
  - Cross-platform (Python)
  - Runs locally
  - Console-based tools
    - **tdm_create_app.py**: Create app skeleton
    - **tdm_build.py**: Build app (and get warnings/suggestions)
    - **tdm_test_interactions.py**: Test specifed app interactions
- TDM **apps** consist of backend and frontend parts
- Backend parts are mainly **declarative** (“what”)
- Frontend part is mainly **imperative** (“how”)

App development philosophy

- All dialogue logic is in TDM, not in the apps
- TDM apps are resources, not programs
- TDM apps provide high-level information
- Benefit: App developers do not need to implement dialogue logic (questions, answers, feedback etc)
- Disadvantage: Sometimes difficult to override default mechanisms

App formalism

- Backend parts:
  - Ontology
  - Grammar
  - Domain
  - Service interface
  - Interaction tests (optional)
  - Mockup service interface (optional)
- Frontend part

Ontology

- Actions
  - *'make_call', 'send_message', 'receive_call', 'receive_message'*
- Sorts
  - *'contact'*
  - Built-in sorts: integer, float, string, boolean
- Predicates
  - *'contact_to_call'* of sort *'contact'*
  - *'contact_to_message'* of sort *'contact'*
- Individuals
  - *'otto'* of sort *'contact'*
  - However: individuals are typically added on the fly, rather than pre-defined

Grammar

- Describes how user and system phrase themselves
- Used for understanding what the user said
- Used for generating system responses
- Consists of mappings between words and semantics
- Examples:
  - “make a phone call” means *request the action 'make_call'*
  - “call X” means *request the action 'make_call' with 'contact_to_call=X' (or with 'contact=X')*

Domain

- Describe how actions are carried out and which information they need
- Describe how questions from user are answered
- Example:
  - *To perform 'make_call', first find out 'contact_to_call'*
  - *To answer 'who_called', query the 'phone' service*
- Also contains various parameters
  - e.g. *when asking 'contact_to_call', say the available options unless they are more than 3*

Service interface

- A.k.a “device”
- Describes how services required by the app are accessed and used
- Consists of
  - Actions (*'MakeCall'*)
  - Queries (*'who_called', 'available_contacts'*)
  - Parameter validators (*'ContactExists')*
  - Entity recognizers (*'ContactNameRecognizer'*)
  - Push notifications (*'IncomingCall' started/ended)*
- Service interface does not implement the actual services – it defers actions, queries etc. to frontend or web API

App formalism

- Formats for backend parts:
  - Python
  - XML (only ontology, domain, grammar; soon service interface)
  - Possible to mix
- Frontend part implemented natively by extending a class
  - e.g. Android Java

Test-driven app development

- Create app skeleton
- Add feature (e.g. making a phone call)
  - Add failing interaction test
  - Validate that new test actually fails
  - Iterate until test succeeds:
    - Modify domain, ontology and/or grammar
    - Build app
    - If fails, re-iterate
    - Run interaction tests
    - If fails, re-iterate

Manual testing

- Text I/O via console
  - Can be useful for quick experiments
  - But automated interaction testing recommended as best practice
- Speech I/O
  - Requires real frontend
  - Required for proper validation of app
  - Good practice: formulate detected failures as interaction tests

Interaction tests

--- setting the time (menu driven)
U> set the time
S> What hour?
U> eleven
S> What minute?
U> ten
S> The time was set to 11 10.

--- setting the time, one-shot
U> set the time to eleven ten
S> The time was set to 11 10.

Simulating failed recognitions

U> what time is it $CHECK
S> Do you want to know the current time?

- Grounding levels:
  - Disregard (too weak confidence; input is ignored)
  - Check (“Do you want to know the current time?”)
  - Acknowledge (“You want to know the current time”)
  - Trust (no feedback required; this is the default)

Service mockup

- Real services can be mocked during interaction testing
- Two methods
  - Mockup service interface
    - Good practice: Extend real service interface and override selected parts
  - Mockup frontend service