Task Completion Platform: A self-serve multi-domain goal oriented dialogue platform

P. A. Crook, A. Marin, V. Agarwal, K. Aggarwal, T. Anastasakos, R. Bikkula, D. Boies, A. Celikyilmaz, S. Chandramohan, Z. Feizollahi, R. Holenstein, M. Jeong, O. Z. Khan, Y.-B. Kim, E. Krawczyk, X. Liu, D. Panic, V. Radostev, N. Ramesh, J.-P. Robichaud, A. Rochette, L. Stromberg and **R. Sarikaya**
Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, USA

Abstract

We demonstrate the Task Completion Platform (TCP); a multi-domain, multi-turn dialogue platform that can host and execute large numbers of goal-orientated dialogue tasks. The platform features a task configuration language, Task Form, that allows the definition of each individual task to be decoupled from the overarching dialogue policy used by the platform to complete those tasks. This separation allows for simple and rapid authoring of new tasks, while dialogue policy and platform functionality evolve independent of the tasks. The current platform includes machine learnt models that provide contextual slot carry-over, flexible item selection, and task selection/switching. Any new task immediately gains the benefit of these pieces of built-in platform functionality. The platform is currently used to power many of the multi-turn dialogues supported by the Cortana personal assistant.

1 Introduction

The aim of the Task Completion Platform (TCP) is to support the rapid development of large numbers of goal-orientated, multi-turn dialogues by simplifying the process of specifying a new task. To achieve this, the definition of individual tasks is separated from the mechanics and policy required to conduct a natural language, goal-orientated dialogue. TCP provides the functionality required to manage dialogues with users, leaving a task author to specify only the information to collect from the user and the interfaces to resources such as data hosted in external services and applications that will execute actions on behalf of the user. The platform is used to power many multi-turn dialogue interactions as part of the Cortana personal assistant.

2 Background

VoiceXML (VoiceXML, 2000) is a standard industry tool used to build dialogue systems. It is typically used to design system-directed dialogues, where the understanding of user input is constrained at each turn, and no opportunity for user initiative. Such dialogues are common in call centre interactive voice response systems, but are of limited utility when more natural interactions are desired, such as the personal assistant dialogues supported by TCP.

A more flexible dialogue management platform is RavenClaw (Bohus and Rudnicky, 2009). RavenClaw systems are composed of a tree structure of agents, with the system at each turn deciding on an execution plan that allows a maximum number of agents to finish their processing. However, most systems built using RavenClaw require custom-built agents; it is difficult to integrate additional tasks into an existing system without having to rebuild the entire tree of agents. In contrast, TCP allows the addition of new experiences with only a configuration file change (as discussed in section 4).

Most similar is ClippyScript (Seide and McDirmid, 2012) which like TCP uses a hierarchical data flow to drive processing (akin to a functional language) as opposed to directed flow control. A key difference is that, in defining tasks in ClippyScript, a dialog act is explicitly tied to a task condition by a rule. Thus, ClippyScript developers directly specify a rule based dialogue policy on a per task basis. In a Task Form (TCP task definition) dialog acts are declared in *association*

with parameters but when to execute them is not explicitly encoded. This allows for separation of tasks and dialogue policies in TCP.

Work also exists on dialogue policy transfer (*e.g.* Wang et al. (2015)) but such work has typically not focused on easy and rapid definition of a diverse set of tasks, as we did in developing TCP.

The main contributions of our work on TCP include: strong separation between task definition and the shared dialogue policy; rapid new task authoring, with the platform providing key machine learning (ML) driven dialogue capabilities such as state tracking, dialogue policy learning, flexible selection, and LU model reuse, which significantly reduces the burden on individual task developers; and benefits to users by offering many multi-turn, mix-initiative dialogue tasks simultaneously.

3 Architectural Overview

The TCP platform supports execution of a variable number of goal-orientated tasks, which can be modified and improved without changing the core platform. The platform has a modular architecture, with the dialogue management process separated into discrete units which can also be independently updated and improved. Figure 1 presents the core platform modules, loosely grouped as: initial processing of input, dialogue state updates, and policy execution. Alternative interpretations of input and dialogue state are preserved in parallel for each step of the pipeline, with a final ranking step to select the optimal hypothesis and associated dialogue act.

The input to the system is either typed text or transcripts from an Automatic Speech Recognition (ASR) system, in the form of N-best hypotheses or confusion networks. The input is processed by a Natural Language Understanding (NLU) module. Several alternate NLU modules can be used in the platform, *e.g.* Deoras and Sarikaya (2013). A typical arrangement is described in Robichaud et al. (2014). To support multiple tasks across multiple domains, a collection of NLU models are executed in parallel, each determining the most likely *intent* and set of *slots* within their domain. A slot is defined as a container that can hold a sequence of words from the user input, with the label indicating the role played by that text in a task or set of tasks. Together, the detected intent and slots form the *semantic frame* for that domain. NLU models are contextual (Bhargava et al., 2013; Liu et al., 2015), taking into account the state of any task currently in progress.

On each turn, the dialogue state is updated taking into consideration the multiple NLU results. *Slot Carry Over (SCO)* (Boies et al., 2016) does contextual carry-over of slots from previous turns, using a combination of rules and ML models with lexical and structural features from the current and previous turn utterances. *Flexible Item Selection* uses task-independent ML models (Celikyilmaz et al., 2014; Celikyilmaz et al., 2015) to handle disambiguation turns where the user is asked to select between a number of possible items. The *Task Updates* module is responsible for applying both task-independent and task-specific dialogue state updates. The task-dependent processing is driven by a set of configuration files, Task Forms, with each form encapsulating the definition of one task. Using the Task Forms, this module initiates new tasks, retrieves information from knowledge sources and applies data transformations (*e.g.* canonicalization). Data transformations and knowledge source look ups are performed using *Resolvers*, as shown in section 4.

Dialogue policy execution is split into task-specific and global policy. The *Per Task Policy* consists of analysing the state of each task currently in progress, and suggesting a dialog act to execute. The output of the module is a set of dialogue hypotheses representing alternative states or dialog actions for each task in progress. The number of hypotheses does not grow uncontrollably; depending on the state of the dialogue during the previous turn and the current input, only a small proportion of all the defined tasks will be active in any one turn.

The output dialog hypotheses are ranked using *Hypothesis Ranking (HR)* (Robichaud et al., 2014; Khan et al., 2015; Crook et al., 2015), which generates a ranked order and score for each hypothesis. This acts as a pseudo-belief distribution over the possible dialogue/task states. *Hypothesis Selection (HS)* policy selects a hypothesis based on contextual signals, such as the previous turn task, as well as the rank order and scores. The HS policy may select a meta-task dialog act, such as asking the user to specifically select a task when two or more tasks are highly ranked, or adding an implicit confirmation when the top hypothesis has low confidence. The

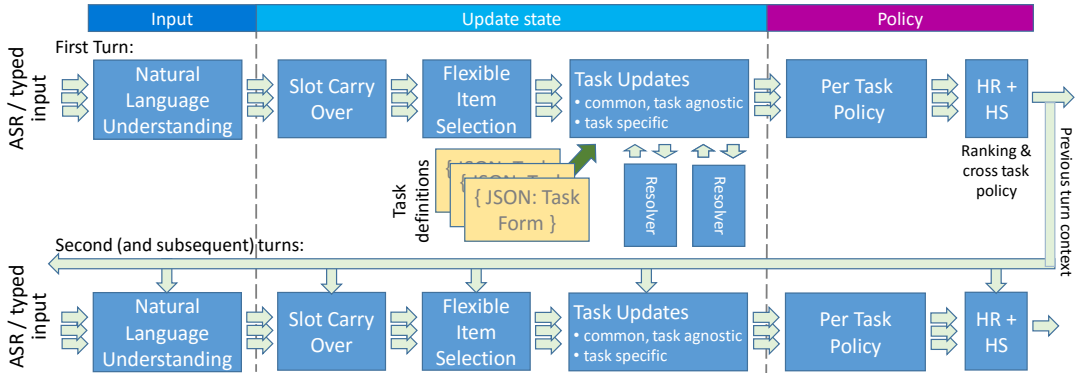

Figure 1: Functional modules of the TCP architecture

selected dialog act is rendered to the user and the associated dialog hypothesis informs the next turn.

4 Task Form Language Definition

We define a task *parameter* as a container that holds knowledge items required to complete a task. These knowledge items can be: concrete entities *e.g.* a particular restaurant in a city; canonicalized attributes of an item, *e.g.* 'small', 'middle' or 'large'; concepts, 'delivery' or 'take-out'; or labels that are indexes to a further knowledge source. Knowledge items are typically retrieved from some knowledge source, *e.g.* a database or knowledge graph, but may be captured directly in code, with NLU slots as input to the lookup or resolution process.

There is a many-to-many relationship between NLU slots and the knowledge items (values) that fill a parameter. Dependent on the run-time results of resolution, additional dialog acts may be required to obtain further user input and refine the resolution. *E.g.* if the NLU slots **[place_name:Starbucks]** and **[absolute_location:Seattle]** are used to index a knowledge source and extract values to fill the task parameter **DeliveryAddress**, the result is likely to be a *list* of Starbucks locations in Seattle. If the parameter specifies that it must hold a single value, further dialog acts will be used to narrow the results. These dialogue policy decisions could not be made by examining the slots in isolation of the resolved values.

The Task Form language allows for defining tasks independently of the dialogue policy executed by the platform. Each task is represented as a set of triggers defining under what conditions task execution should begin, a set of parameters defining what information should be collected during the dialogue,

```
{"TriggerName": "PizzaOrdering",
 "Intent": "order_pizza",
 "TriggerOnlyOnTriggerQueries": false,
 "TriggerQueries":
  ["i wanna order a pizza",
   "i want to order a pizza",
   "order domino's pizza",
   "hey cortana, get me a pizza",
   ... ]}
```

Figure 2: An example definition of a task trigger.

and a set of dialog acts defining what is presented to the user. Additional structural information about a task can be captured through the use of validation conditions (not shown here).

A *task trigger* defines the conditions required for the execution of a task to begin. These conditions are represented in terms of language understanding results, specifically combinations of domains, intents, and slots (presence or absence). Additionally, each trigger may specify a fixed list of utterances which should trigger the task, as shown in figure 2.

Each *task parameter* defines a container for information required for task execution, *e.g.* figure 3. The information may be collected directly from the user, inferred during the execution, or mixed. The definition of each parameter specifies how the value of the parameter should be produced, together with what dialog acts are used to solicit relevant information. A 'RequiredIf' statement marks the parameter as conditionally optional given other parameters. This encoding of parameter relationships allows for expressing a richer set of task structures.

The value of a parameter is provided by an associated piece of code, a *resolver*. The actual resolver code is implemented outside the Task Form. In the Task Form a resolver definition (optional for

```
{ "ParameterName": "DeliveryAddress",
 "Description": "delivery address",
 "Type": "Location",
 "RequiredIf": "DeliveryOption == 'Delivery' ",
 "ResolverInvocation":
  {"Resolver": "Platform.PlacesResolver",
   "ResolverSlotTags": [ "absolute_location", "place_name", "place_type" ],
   "ResolverOutputParameterName": "PlaceParameter" },
 "DialogActs": {
 { "MissingValue": "DeliveryAddressMissingValueDialogAct",
   "Disambiguation": "DeliveryAddressDisambiguationDialogAct" } },
```

Figure 3: An example definition of a parameter.

resolvers provided by the platform) contains only a reference to the code assembly, class name, and a list of slot tags and parameters which it can process. The parameter definition contains a resolver invocation block which lists a subset of slot tags and input parameters that should actually be used as input to the resolver during the task execution; this allows resolvers to be implemented more generically and be reused across multiple parameters and tasks.

As part of the execution of a task, the system may take one or more dialog acts before the value of a parameter is considered "filled". Allowed acts include:

- **MissingValue** - ask the user for input required to populate the parameter (plus a variation that presents suggested values to the user);
- **NoResultsFollowup** - prompt to change information as no results were found (plus a variation that presents suggestions);
- **Disambiguation** - ask the user to select the parameter value from a list;
- **ImplicitConfirmation** - implicitly confirm the newly filled parameter as part of the next turn;
- **Confirmation** - ask the user to confirm the parameter value the system computed;
- **ConfirmationFailure** - ask the user to provide new input if they rejected a confirmation act.

A *dialog act definition* captures the information that should be presented to the user when that dialog act is taken by the system. This information includes: a prompt to be read out, a list of strings to be shown on the screen, as well as hints to prime the NLU during the next turn of the conversation. Default definitions are used for any missing sections in each dialog act definition. Figure 4 shows an example of a dialog act definition, encoding how the user should be prompted to provide the missing value of the *DeliveryAddress* parameter.

```
{ "DialogActName": "DeliveryAddressMissingValueDialogAct",
 "Response":
 { "Prompts":
  [ "Where do you want the pizza delivered?",
    "Where do you want it delivered?",
    "Where should I send your pizza?" ],
  "SpokenPrompts":
  [ "Now, where do you want your pizza delivered?",
    "Okay, where do you want it delivered?",
    "Now tell me, where should I send your pizza?", ] } }
```

Figure 4: An example definition of a dialog act.

5 Demo Outline

We plan to showcase the capabilities of TCP, highlighting in particular the breadth of the platform and the agility of task development.

The platform is capable of executing multiple tasks using the same underlying policy modules, thus allowing for tasks to be developed separately from the policy definition. To this end we will show the platform supporting a conversational agent capable of setting reminders, ordering pizzas, and reserving movie tickets, restaurant tables and taxis (where these scenarios are hooked up to real third party services like Domino's, OpenTable, Uber, *etc.*). Each task is defined by a Task Form. Users can interact with the system through natural conversation and take initiative at any point, *e.g.* to cancel a task in progress, provide information out of turn, or change previously-provided information.

Many changes to a task can be done simply by manipulating its Task Form definition. To illustrate this, we will demonstrate some simple modifications to an existing task, such as changing the task triggers, adding a new parameter, modifying some of the conditions set on parameters, and redefining some of the dialog acts used during task execution.

6 Conclusion

We demonstrated the Task Completion Platform, an extensible, mixed-initiative dialogue management platform targeting goal-directed conversations. The platform supports executing multiple tasks. Each task is defined primarily in terms of extensible NLU models and a single configuration file, thus separating task definition from the system-wide dialogue policy. Tasks can be added or modified without requiring a system rebuild. Future work includes extending the platform by allowing multiple tasks to be concurrently in progress, either through nesting or interleaving of tasks.

References

A. Bhargava, A. Celikyilmaz, D. Hakkani-Tur, R. Sarikaya, and Z. Feizollahi. 2013. Easy contextual intent prediction and slot detection. In *Proc. ICASSP*, May.

D. Bohus and A. Rudnicky. 2009. The RavenClaw dialog management framework: Architecture and systems. *Computer Speech and Language*.

D. Boies, R. Sarikaya, A. Rochette, Z. Feizollahi, and N. Ramesh. 2016. Using sequence classification to update a partial dialog state (United States patent application 20150095033, filed 2013).

A. Celikyilmaz, Z. Feizollahi, D. Hakkani-Tur, and R. Sarikaya. 2014. Resolving referring expressions in conversational dialogs for natural user interfaces. In *Proc. EMNLP*, October.

A. Celikyilmaz, Z. Feizollahi, D. Hakkani-Tur, and R. Sarikaya. 2015. A universal model for flexible item selection in conversational dialogs. In *Proc. ASRU 2015*, December.

P. A. Crook, J.-P. Robichaud, and R. Sarikaya. 2015. Multi-language hypotheses ranking and domain tracking for open domain dialogue systems. In *Proc. Interspeech*, September.

A. Deoras and R. Sarikaya. 2013. Deep belief network Markov model sequence classification spoken language understanding. In *Proc. Interspeech*, August.

O. Z. Khan, J.-P. Robichaud, P. A. Crook, and R. Sarikaya. 2015. Hypotheses ranking and state tracking for a multi-domain dialog system using ASR results. In *Proc. Interspeech*, September.

C. Liu, P. Xu, and R. Sarikaya. 2015. Deep contextual language understanding in spoken dialogue systems. In *Proc. ASRU*, December.

J.-P. Robichaud, P. A. Crook, P. Xu, O. Z. Khan, and R. Sarikaya. 2014. Hypotheses ranking for robust domain classification and tracking in dialogue systems. In *Proc. Interspeech*, September.

F. Seide and S. McDirmid. 2012. ClippyScript: A programming language for multi-domain dialogue systems. In *Proc. Interspeech*.

VoiceXML. 2000. VoiceXML version 1.0. https://www.w3.org/TR/voicexml/.

Z. Wang, Y. Stylianou, T.-H. Wen, P.-H. Su, and S. Young. 2015. Learning domain-independent dialogue policies via ontology parameterisation. In *Proceedings of SIGdial*.