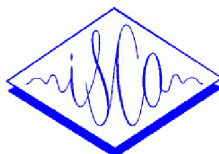
ISCA Archive
http://www.isca-speech.org/archive



INTERSPEECH 2012
ISCA's 13th Annual Conference
Portland, OR, USA
September 9-13, 2012

ClippyScript: A Programming Language for Multi-Domain Dialogue Systems

Frank Seide and Sean McDirmid

Microsoft Research Asia, Beijing, P.R.C.
`{fseide,smcdirm}@microsoft.com`

Abstract

The past year has witnessed the revival of spoken dialogue systems. As these systems become multi-domain and more ubiquitous, the efficient scripting of dialogues increases in importantance. As of today, statistical approaches to dialogue control are not yet feasible; the problem remains quite firmly one of manual coding.

This paper describes an object-oriented programming language that we have christened ClippyScript, which is aimed at the rapid manual scripting of multi-domain dialogue systems. Only four main keywords—MODULE, SLOTS, GRAMMAR, and ACTIONS—plus a concept of *focus* provide the necessary abstractions for language understanding, dynamic grammars, hierarchical slot filling, multiple domains, access to data services, and performing of the dialogue goal. The language's expressive power is further boosted by the ability of embedding code snippets in a high-level programming language (C#).

Index Terms: spoken dialogue systems, scripting languages

10.21437/Interspeech.2012-77

1. Introduction

Spoken-dialogue systems are on a firm path to ubiquity given the recent successful roll out of such a system on a major smart phone as a prominent means of user interaction. Eventually, third-party applications will be voice-enabled as well. How can non-speech specialists, who develop these apps, develop high-quality spoken-dialogue systems in a rapid manner?

Ideally, we could automatically derive dialogue systems from any program's existing UI, possibly annotated with hints to indicate a dialogue policy. This does not exist today, and may never, due to the fundamentally different nature of those UIs and human-like dialogues. Alternatives are statistical approaches for learning dialogue systems from observing human-human dialogues, like POMDP [1], but, in addition to data-scarcity problems, they are not quite ready yet for the complex systems that we are concerned with. As a pragmatic approach, we focus on easing the *manual crafting* of a complete dialogue system.

So what are the right *abstractions* for expressing a spoken dialogue system? Abstractions provide constraints. Too few constraints will lead to inefficient development and low-quality systems; e.g., today's speech SDKs provide only rudimentary facilities, like start/stop recognition and switching grammars, while leaving everything else to the programmer. Too many constraints, however, may prevent programmers implementing what they desire, or force them into quirky workarounds.

Next, what is the right way of *coding* to our chosen abstractions? Historically, abstractions often start as libraries accessible to existing languages, but over time can be promoted to language-level constructs. For example, early GUI programs were written in procedural languages such as Pascal and C++, while today's programming languages, such as C#, support them more directly with event mechanisms and delegates.

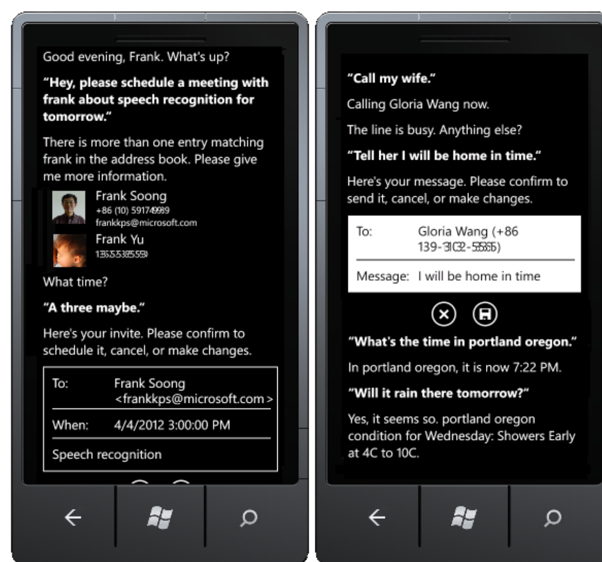


Figure 1: *Personal assistant prototype written in ClippyScript.*

This paper proposes a domain-specific programming language (DSL) aimed at providing the right abstractions for programmers to manually but rapidly script multi-domain, mixed-initiative, slot-filling spoken-dialogue systems with no automatic optimization. Our language supports spoken-language parsing based on probabilistic grammars, a hierarchical slot resolution architecture, micro and macro control flow through a predicate-action based rule system, and integration with system and data services, in a modular architecture aimed at code reuse.

The language is kept simple by strictly limiting its purpose. Like in the classic Yacc compiler generator, code snippets in a "foreign" general-purpose language (C#) can be embedded in the script wherever suitable. We call the language "codename ClippyScript" as a tribute to an earlier attempt at intelligent assistants [2], which otherwise has nothing to do with this work.

ClippyScript incorporates several concepts of prior DSLs, including the declarative predicate-action rule concept of Philips HDDL [3], which is also found in the Information-State Update (ISU) model [4], and the slot hierarchy of ActiveOntologies [5]. Compared to prior DSLs, ClippyScript avoids the syntactic overhead of VoiceXML [6] and models a hierarchical data flow while RavenClaw [7] is organized by control flow. ISU has a similar rule engine that is, however, applied to abstract rules based on "dialogue moves," leaving domain rules to code, while ClippyScript makes no distinction.

Fig. 1 shows screen shots of a live prototype of a personal assistant on a mobile phone, encompassing about ten domains such as scheduling, messaging, or weather, that are implemented in ClippyScript to validate its language design.

2. Execution Model

In this section, we introduce our computational model for dialogues, handling of multiple domains, and system integration.

2.1. A Hierarchy of Slot Functions

The central concepts in ClippyScript are *slots* and *slot functions*. Like [5], we model a dialogue as a hierarchy of slots where data flows upwards from spoken inputs to a top-level *root slot*.

Every slot in the hierarchy has an associated *slot function* that realizes the slot's value from its inputs, which in turn are slots. The root's slot function is the operation the user intends to execute, such as scheduling a meeting. A dialogue is successful when this function is executed. The root's inputs are intermediate *belief slots* that act as the arguments to its slot function.

Each belief slot, in turn, realizes its value through an associated slot function whose arguments are either further down-level belief slots or bottom-level transient *input slots* that represent the current utterance and are filled by the spoken-language parser. Belief slot functions typically either aggregate their inputs into a more complex data object, or combine inputs to resolve ambiguity. Both can involve complex operations such as database lookups and web-service calls; such as searching the address book for a name or finding available flight connections.

The structure above defines a tree-like dependency graph of slots through which information flows upwards from input slots through belief slots to the root, which gives rise to our *dialogue model of computation*:

> By recursively traversing the graph, compute all inputs to the root slot's function so as to execute it; if this fails because a required input anywhere in the graph is unavailable, pause the execution and prompt the user for the missing information.

This model is sufficient for implementing mixed-initiative dialogues. Our specific implementation of this model involves two additional slot types, *output* and *context slots*, that we describe later.

2.2. Extension to Multiple Domains

To handle multiple application *domains* and *intents*, we extend our dialogue model with multiple roots, one for each domain intent. (We sometimes group similar intents into a single root and distinguish them within the root slot function.) Multiple domains often share the same inputs; e.g., "send e-mail" and "schedule a meeting" both require a "person" object. The corresponding slot functions can just share these inputs, and the dependency tree becomes a graph. Slot sharing is also a natural mechanism for handing over information between domains.

Unfortunately, multiple domains can cause significant ambiguities for both the spoken-language parser, which is domain-conditioned to maximize accuracy, and the processing of the slot hierarchy. We address this through an *input focus* metaphor known from UI programming: Exactly one root has the input focus at any given time, meaning that only intermediate slot functions and input slots within that root's dependency tree are subject to computation and can receive spoken-language input.

An initial *open-intent phase* occurs when domain and focus are still unknown. This phase is implemented by giving input focus to an *intent-arbiter* slot that, in turn, has all roots as its inputs; i.e., "root slots" themselves are intermediates in the hierarchy. This topology is also useful when considering nested intents; e.g., pick a movie, find a theater, purchase ticket online, and get driving directions. Albeit now inaccurate, we will continue to use the term "root slot" for simplicity.

Focus changes are enacted through slot functions, predominantly by slots that are indicative of intent (domain/intent switch), or after the successful execution of a root slot function to reset the focus to the intent arbiter. Note that since the spoken-language parser is domain-conditioned and only parses for slots defined in a domain, a focus change may lead to a different parse and thus to an update of the input slots.

The notion of input focus, in conjunction with multiple roots, then arms us with a powerful tool for implementing multi-domain dialogues.

2.3. System Integration by Embedding Foreign Code

We designed our model with two features that allow it to closely interact with our chosen operating environment. First, slots have types and can hold any simple or complex object types that are supported by the runtime environment, which in our case is the Microsoft .Net Framework. Data types can also be imported from external libraries; consider a class holding a set of movies showing in nearby theaters complete with show times, user ratings, and thumbnail URLs. Second, expressions written in an existing *external* programming language, in our case C#, can be embedded in ClippyScript. Such embedded "foreign" expressions are directly executable by the runtime environment *while having access to slot values*, and are a crucial part of our design for two use cases: First, they support the construction of compound objects from down-level slot values for use as an up-level slot value; and second, they can access external databases and web services. The alternative to foreign code would be to enrich the dialogue-description language enough to implement these directly, as done in HDDL, but such an enriched language would be costlier to implement and harder to learn.

The full power of ClippyScript and its foreign-code capabilities unfolds when they work hand in hand; e.g., when disambiguating through a web service. Consider resolving a city name into a location id, such as a Yahoo "where-on-earth id" that some relevant web services require as an input. The location-lookup web service might return multiple matches that the user must disambiguate through dialogue scripted in ClippyScript. The dependency tree is the "skeleton" of the dialogue process, while embedded foreign code provides the "meat."

3. Language and Implementation

This section presents how the execution model of Section 2 is actually realized in ClippyScript with abstractions for spoken language parsing, slot functions, predicated actions, modules, context, and code embedding.

3.1. Spoken Language Parsing

Input slots at the bottom level of the slot hierarchy collectively constitute a semantic frame that represents the meaning of the current input utterance. Spoken-language parsing transforms the sequence of input words into that semantic frame. ClippyScript supports bottom-up parsing with hand-written BNF grammars. To handle ambiguous input robustly, we follow [3] in that grammar rules have trainable probabilities; a bigram language model models the overall sequencing of patterns, and a filler model covers words between matching grammars.

ClippyScript's BNF syntax is augmented by rules to compute semantic interpretations. Similar to slots, these interpreta-

tions can be of arbitrary data types, although typically they are primitive types like strings or date/time. Expressions for computing semantic interpretations are written as embedded C# expressions. A special syntax denotes non-terminals that fill input slots. The following example encodes how a slot **amount** is filled with a dollar amount between 20 and 99:

```
SLOTS { INPUT amount : int; }
GRAMMAR
{
  $digit1to9  : int =    one => ((1))   | ...
  $tens20to90 : int = twenty => ((20))  | ...
  $nums20to99 : int
    = $tens20to90 => ((tens20to90))
    | $tens20to90 $digit1to9 =>
                     ((tens20to90 + digit1to9));
  INPUT(amount)
    = $nums20to99 dollars => ((nums20to99));
}
```

A preceding **$** distinguishes non-terminals from terminal tokens. Attribute expressions of the form **=> ((·))** contain embedded C# code that can access right-hand side non-terminal values as variables named after the non-terminals. **INPUT(·)** indicates a rule that fills a slot; in this case the **amount** slot.

The clean and compact BNF notation makes it convenient to write grammars directly inside the dialogue script close to other related elements of the dialogue; as opposed to, for example, VoiceXML, where grammars are often in external resources. As another powerful mechanism, grammar rules can also be loaded from slots whose values are constructed dynamically by slot functions, which allows to parse context-dependent items like names in the user's address book or items returned by a web service. Finally, ClippyScript also supports statistically modeled "free-form" segments of uninterpreted word sequences, like the body of a text message, or phrases that are interpreted directly by an underlying data service, such as a street address that is passed into a map application.

3.2. Slot Functions

When a new user utterance is received, all input slots are replaced by those representing the new input. This update then triggers upward data flow in the slot hierarchy by means of *slot functions*. Our computational model of dialogue consists of data flowing upwards through the slot hierarchy, from input slots to the root. This flow is realized by evaluating the value of each slot in the hierarchy by applying its slot function whose parameters are the values of down-level slots, which in turn are computed by applying this process recursively, down to input slots that are filled by the spoken-language parser at each turn.

Slot functions are the central element of our computational model and are used for many purposes, including aggregating input slots, performing external database lookups, and even executing the final dialogue goal. Slot functions also manage (a) high-level dialogue control flow through their ability to change input focus; and (b) within-domain low-level control flow by determining which inputs are missing and prompting the user for disambiguation or additional information.

A slot's slot function is defined by the *totality* of all predicate-action rules affecting the slot, as we will discuss next.

3.3. Predicate-Action Rules Realize Slot Functions

Unlike high-level flow, *micro state* (within-domain low-level control flow) cannot easily be handled by *explicit* state transitions. Instead, this process is encoded in ClippyScript through predicate-action rules. A *predicate* defines a micro state, and an *action* an appropriate operation to take when in that state. An action consists of one or more statements (predominantly slot assignments). Dialogue computation consists of repeatedly determining the first matching predicate and executing its action.

In addition to updating belief slots, actions can also assign to *output slots* that enable user interaction. Once an action assigns to one or more output slots, rule processing is put on hold and the output slots are sent to the client that then communicates them to the user and requests input.

The following is an excerpt of predicate-action rules for making a phone call. The excerpt demonstrates how to disambiguate a name in case of multiple matches in the address book. We assume that the intent ("call") has been detected, and the name of the person to call fills the input slot **name**:

```
SLOTS
{
  INPUT  name : string;    // "call" <name>
  INPUT  displayselected : int; // item touched
  BELIEF hits : List<App.Contact>; // matches
  BELIEF contact : App.Contact; //resolved name
  OUTPUT prompt : string;  // text to show
  OUTPUT display : object; // graphics to show
  OUTPUT action : object;  // action to perform
}
ACTIONS
{
  MAKE (hits : name) :          // look up name
    hits <- ((AddressBook.Search(name)));
  MAKE (contact : hits, displayselected) ://tap
    contact <- ((hits[displayselected]));
  MAKE (contact : hits) :       // disambiguate
  {
    ((hits.Count == 0)) :       // no match
      prompt <- "No such entry.";
    ((hits.Count == 1)) :       // one match
      contact <- ((hits[0]));
    ((hits.Count == 2)) :       // two matches
      prompt <- ((hits[0]+" or "+hits[1]+"?"));
    ((hits.Count > 2)) :        // multiple
      prompt <- (("Which " + name + "?")),
      display <- ((new App.ShowNames(hits,
                          "displayselected"))),
  };
  MAKE (contact) :              // no name given
    prompt <- "Call who?";
  MAKE (action : contact) :     // place the call
    prompt <- (("Calling " + contact.fullname)),
    action <- ((new App.Call(contact.phoneno))),
    SWITCHTO (intentarbiter);
}
```

MAKE is a predicate that tests the availability and recency of slot function parameters. The colon in **MAKE** denotes a dependency as in a makefile; e.g., **MAKE (hits : name)** is **true** if the **name** slot has been filled and the **hits** slot is empty or outdated with respect to **name**; if the rule matches, its action places all matching address-book entries into the **hits** slot as a C# list.

Rule processing then starts over but the **MAKE** rule for **hits** will no longer match since **hits** is up to date. If the **hits** slot was bound to a unique entry in the address book (one match), then the slot **contact** is resolved right away; otherwise the system prompts back by filling an output slot. A display of names is created on the client in the case of many matches. To allow for multi-modal (touch) selection, the object is given the name of an input slot (**displayselected**) that will be set to a name's index when the user taps on the name. A predicate-action testing for the **displayselected** slot then resolves the contact.

Once the **contact** slot is resolved, the system proceeds to place the call by sending an appropriate **action** object with the phone number to the client. Notice that the **contact** slot has a C# data type with member variables that can be accessed by the embedded C# expression. Finally, because this intent has successfully been completed, a **SWITCHTO** statement changes the input focus back to the "intent arbiter," meaning the dialogue is now open again for any domain.

This example involves three output slots. ClippyScript itself is agnostic as to the meanings of output slots, which are

only interpreted by the client. Another use of output slots that is not shown above is to include a row of buttons that allow the user to accept or cancel an operation without having to speak.

The example also demonstrates several other things. First, ClippyScript does *not* define the slot hierarchy explicitly as a graph structure. Rather, the slot hierarchy is defined implicitly through **MAKE** predicates. Second, a **MAKE** rule not always leads to an update—it can also lead to a disambiguation. Third, slots can be filled through multiple expressions.

3.4. Modules and Code Reuse

Although our computational model is designed around a hierarchy of *slots* realized through **MAKE** predicates, ClippyScript itself is built around a hierarchy of *modules* that improve clarity and foster code reuse. Modules can group slots together and recursively "derive" from other modules, through mixin-style linearized multiple inheritance found in some in object-oriented languages like Scala. A top-level module typically represents a domain that implements one or more related intents, while deriving down-level modules typically contain everything related to a certain slot; i.e., the slot itself, its associated grammars, resolution rules and dialogue fragments, and also helper variables that themselves are considered slots.

In practice, the module hierarchy closely mirrors that of the slot hierarchy, but modules can also just hold basic grammars for use at multiple places, like for generic numerals. In contrast, ActiveOntologies [5] expresses a similar slot hierarchy by explicitly declaring the dependencies between slots and their parameters in a graphical manner. We chose instead to separate the module and slot hierarchies for added flexibility. Although it is good practice to associate one module with one slot, decoupling them is useful for two reasons: First, for tiny domains, we avoid unnecessary syntactic clutter; and second, some slots can involve more complicated update rules, such as being fed from multiple alternative parameters, or may require the use of helper slots like `displayselected` in the above example; such rules are best grouped together. As an example, consider the calendar module declaration:

```
OUTPUT MODULE calendar : io, calendar_intent,
             datetime, contact, about, location
{
  GRAMMAR { ... }
  INPUT ACTIONS { ... }
  BELIEF ACTIONS { ... }
  OUTPUT ACTIONS { ... }
}
```

The calendar module contains (imports) everything defined in the six modules that it inherits from. In fact, none of the inputs for meeting requests are defined in the calendar module itself; they all come from respectively named inherited modules.

The module hierarchy is used directly in two ways. First, our concept of *input focus* is applied at the level of *modules*, rather than slots themselves. Second, predicate-action rules are processed depth first within the module hierarchy. Beyond this, the module hierarchy fulfills three primary purposes. First, it closely mirrors the slot hierarchy and thus describes it. Second, it allows the sharing of code like basic grammars. Finally, when modules are shared across domains, the same instance of a slot becomes visible to multiple domains. This enables cross-domain dialogues such as (1) making a phone call to someone in the address book and (2) getting driving directions to that person's home.

Let us revisit a module-related aspect of spoken-language parsing. The module example above has its own **GRAMMAR** section because the calendar domain involves expressions where each slot by itself would be too brittle to detect by its own grammar; e.g., in "remember to buy milk," 'to' alone is too weak an indicator for a reminder text. We allow for the assignment of multiple slots from a single grammar rule to improve parsing; consider:

```
GRAMMAR
{
  $remindexpr
    = remind me | remember | set reminder;
  $freetext : string = ... ;   // any sequence
  INPUT(intent,about) = $remindexpr $freetext $
    => intent <- ((Intents.SetReminder)),
       about <- ((freetext));
}
```

3.5. Static and Dynamic Context

Besides input, belief, and output slots, which have limited lifetime, *context slots* store persistent information, such as user information like their name, location, even their entire address book, which is associated dynamically generated grammar for the parser. Context slots also hold long-span references for anaphora resolution ('her,' 'there'). Fig. 1 contains several examples that are realized through context slots.

3.6. System Integration: Embedding Code

Embedded C# expressions are handled as follows. When a dialogue script is loaded and parsed, all embedded expressions are collected and converted into C# "code-behind" source code that is compiled and evaluated on the fly through C#'s reflection mechanisms.

3.7. Experience: Prototyping a Personal Assistant

The ClippyScript language and engine were co-developed with a prototype of a spoken-dialogue based personal assistant (Fig. 1). ClippyScript was instrumental to the efficient scripting of its ten domains, which constitute a total of about 380 rules, of which 270 are one-liners. Most domains' initial dialogue-policy implementations took less than a day to code.

4. Conclusion

We have presented ClippyScript, a domain-specific programming language aimed at efficient scripting of multi-domain, mixed-initiative, slot-filling spoken dialogues. ClippyScript abstracts the problem into four basic primitives: *slots*; hierarchical slot functions defined through *predicated actions*; *input focus*; and the object hierarchy of *modules*. Their composition, in combination with the ability of embedding spoken-language grammars and code snippets in a high-level programming language (C#), makes ClippyScript a powerful tool for rapid scripting of dialogue systems. This has been validated by a prototype of a spoken-dialogue based personal assistant.

5. References

[1] J.-D. Williams and S. Young, "Partially observable Markov decision processes for spoken dialog systems," Computer Speech and Language 21 (2007).

[2] `http://en.wikipedia.org/wiki/Office_Assistant`

[3] H. Aust, M. Oerder, F. Seide, and V. Steinbiss, "The Philips automatic train timetable information system," Speech Communication, Vol 17, No. 34, Nov. 1995.

[4] D. Traum *et al.*, "The Information State Approach to Dialogue Management." Current and New Directions in Discourse and Dialogue, pp. 325–353, Ed. J. v. Kuppevelt *et al.*, Kluwer, 2003.

[5] D. Guzzoni, C. Bauer, and A. Cheyer, "Modeling Human-Agent Interaction with Active Ontologies," Artificial Intelligence SS-07-04, 52-59, 2007.

[6] `http://www.voicexml.org`

[7] D. Bohus *et al.*, "RavenClaw: Dialog Management Using Hierarchical Task Decomposition and an Expectation Agenda," Eurospeech, 2003.