Adapting Dialog Call-flows for Pervasive Devices

Nitendra Rajput, Amit Anil Nanavati

IBM India Research Lab
Hauz Khas, New Delhi - 110016.
{rnitendra, namit}@in.ibm.com

Abhishek Kumar, Neeraj Chaudhary

Dept. of Electrical Engg.
I.I.T., Hauz Khas, New Delhi - 110016.
{eeu02462, eeu02474}@ccsun50.iitd.ac.in

Abstract

There is an increasing variety of pervasive devices in use today. More and more applications are being supported on such devices. This requires device-specific application adaptation. We address the problem of speech application adaptation by dialog call-flow reorganisation for pervasive devices with different memory constraints. Given a dialog call-flow $C$ and device memory size $M$, we present deterministic algorithms that alter $C$ to create $C_n$, that fits $M$ by increasing the number of questions and splitting the underlying grammar. We can split a grammar by exposing the intermediate non-terminals in the grammar. The following observation forms the cornerstone of this paper: An and-grammar can be split “horizontally”, and an or-grammar can be split “vertically” into its components to reduce the memory requirement of the call-flow, at the expense of increasing the number of prompts (questions). We present algorithm G-split. explain its implementation with example call-flows authored in VXML containing SRGS grammars.

1. Introduction

Users are increasingly accessing remote applications from and running local ones on their mobile devices. Since mobility is a major factor, voice-enabled applications have acquired even more importance. More and more applications are becoming voice-enabled. Pervasive devices differ from desktop computers in two fundamental ways. One, they occur in various sizes with vastly differing capabilities, and by virtue of mobility, are not always connected to the network. This combination gives rise to interesting challenges and possibilities.

We are interested in device-specific adaptation of speech applications. Traditionally, speech applications run on a remote server, and several client-server interactions take place in the course of a dialog. Server-side processing intensive systems suffer from the vulnerability of server bottlenecks. A typical approach to alleviate such bottlenecks and achieve scalability is to offload processing to the client side to the extent possible [1]. The evolution of Javascript in the context of the Web is an example. Second, for pervasive devices, server connectivity comes at a cost and is not always robust. A client-server model incurs transmission costs, and is prone to transmission errors, which could result in degraded speech recognition accuracy. The use of speech compression for reducing transmission costs introduces other complications [2]. In order to circumvent such problems, speech recognition on the client device offers a viable alternative [3]. Together, these factors make a compelling case for disconnected, client-side processing of dialog call-flows [4].

While the available memory size on devices is increasing, so is their ability to support more and more complex conversational systems. Conversational systems range from single-word-based-recognition to phrase-recognition to complex-grammar recognition to large-vocabulary-recognition coupled with NLU, in the order of increasing memory requirements. The state of the art advanced NLU tasks such as “how-may-I-help-you” type of tasks, which provide much better user experience) require more memory than is typically available on laptops ($\approx 512\text{ Mb}$). Smaller handheld devices have far less memory than a conventional laptop, therefore it becomes necessary to adjust the complexity of a conversational system for different devices.

We investigate the problem of dialog call-flow reorganisation for pervasive devices with memory constraints. The crux of the reorganisation lies in altering the memory requirement of the underlying grammar [5]. There are two types of grammars, and-grammars and or-grammars. One can continually split a grammar, till either the resulting grammar size can be supported by the device or it is no longer possible to split it further. A grammar consists of non-terminal symbols which may not have corresponding prompts that are used in user interactions. The essence of splitting lies in exposing these intermediate non-terminals with an explicit prompt. We present a deterministic algorithm, G-split, which alters a given call-flow (sequential, tree-type, or hybrid) to fit the memory constraint. The G-split takes as input a VXML dialog containing SRGS (Speech Recognition Grammar Syntax) grammars [6] and a memory size $M$. Among other things, it “exposes” an intermediate non-terminal by converting it to a “root” node. VXML and SRGS parsers parsers are used to parse the input dialog to build a call-flow. The output of the G-split is converted back to a VXML dialog.

Section 2 starts with some preliminaries and presents and G-split. Section 3 details the implementation and demonstrates the functioning of G-split with an example. Section 4 concludes the paper.

2. Splitting Dialog Call-flows

This section starts with some preliminary definitions, followed by a description of the main ideas behind G-split. We then detail algorithm G-split which splits a dialog call-flow until it fits a given memory constraint $M$, or stops if it is unable to do so.

2.1. Preliminaries

An and-grammar of size $n$ is of the form $G := g_1g_2\ldots g_n$, where each component grammar $g_i$ may be a terminal or a non-terminal. An or-grammar of size $n$ is of the form $G := g_1\mid g_2\ldots\mid g_n$. There is a prompt associated with the answer grammar $G$. Splitting an and-grammar $G$ results in “exposing” two or more $g_i$s by associating a prompt with each $g_i$. 

...
Figure 1: Splitting an and-grammar. A grammar \( g \) comprising of \( g_1 \) and \( g_2 \) can be split into two grammars \( g_1 \) and \( g_2 \), reducing the number of answer choices from \( |g_1| \times |g_2| \) to \( \max(|g_1|, |g_2|) \).

To find an answer for \( G \) is equivalent to finding the answers to each \( g_i \) and collating them. Splitting an or-grammar \( G \) results in a regrouping (subset formation) of the \( g_i \)s into new non-terminal symbols \( G_j \)s. This results in the following grammar: \( G := G_1 | G_2 | \ldots | G_k \) such that \( G_i \cap G_j = \emptyset; \forall g_i, \exists G_j \ni g_i \in G_j \) and \( k < n \). The idea behind splitting is to reduce the number of choices for a prompt at a given level, so that the entire set of choices can fit a given memory constraint. As a result of splitting, at least 1 additional prompt is generated. Informally, an and-grammar is split horizontally into its components, while an or-grammar is split vertically into many levels. Thus splitting an and-grammar results in new prompts for its components, while splitting an or-grammar results in new prompts for each additional level. An atomic grammar is one that cannot be split any further. \( G := g_1 \) and \( G := g_1 | g_2 \) where \( g_i \)s are terminals, are atomic grammars. Figures 1 and 2 show the effects of splitting an and-grammar and an or-grammar respectively.

**Observation 1** In the case of an and-grammar of size \( n \), splitting introduces a maximum of \( n \) new prompts (the old prompt is discarded). The length of the call-flow increases by \((n-1)\) levels. For an or-grammar of size \( n \), splitting may introduce as many as \((n-2)\) new prompts, and at most \( \log(n)\) levels in the tree. Since these operations are all \( O(n) \), this ensures that repeated splitting still terminates in polynomial time.

2.2. Reorganisational Constraints

Reorganisational constraints are of two types. One, that insist that a certain dialog be split, and two, that forbid a certain dialog from being split. The first set must-split is a set of dialogs which must be split. The second set dont-split is a set of dialogs which should not be split. One reason for insisting on splits might be to improve recognition accuracy. One reason for insisting on not splitting might be that it makes logical sense to keep things together (usability), for example credit card number and expiry date. Reorganisational constraints, which may be specified manually, thus provide a mechanism to incorporate various practical considerations and constraints to improve the overall usability and performance of a dialog call-flow.

2.3. G-split

We present algorithm \( G \)-split that alters a call-flow \( C \) to fit a given memory constraint \( M \), within reorganisational constraints. \( G \)-split is described below:

\( G \)-split

1. input: reference sequential call-flow \( C \), memory size \( M \).
2. output: altered sequential call-flow \( C'_m \).
3. Construct a graph \( G(V, E) \) as follows:
   (a) Represent all dialogs by vertices labelled \( \{1, \ldots, n\} \).
   (b) Let \( C'_m = C \).
   (c) for each vertex \( i (1 \leq i \leq n) \)
   (d) if "must-split\( g_i \)" \& \( m(g_i) > M \), split\( (g_i, C'_m) \).
   (e) if must-split\( g_i \), split\( (g_i, C'_m) \).
4. output \( C'_m \).
5. split\( v, C'_m \)
   (a) if \( \text{and}(v) \) \( \| v \) is an and-grammar
   (b) for \( 1 \leq i \leq n \) \( \| v \) has \( n \) components
      i. if \( m(g_i) > M \), split\( (g_i, C'_m) \).
      ii. if \( \sum m(g_i) > M \) split at (i-1); else continue.
   (c) else for all terminals in \( v \) \( \| v \) is an or-grammar
   (d) \( S = \left\{ \sum m(g_i) / M \right\} \| S \) groups
   (e) for each unexposed non-terminal \( g_i \) in \( v \)
   (f) if \( \text{non-terminal} m(g_i) > M \) split\( (g_i, C'_m) \); else expose \( g_i \).

According to step 3(d) and 3(e), split is called when there is a "must-split" or the memory requirement exceeds \( M \). The split...
function (step 5) handles both and-grammars and or-grammars. Step 5(a)-(b) address the and-grammar. When an individual component exceeds the memory constraint \( M \), split has to be called recursively to break it further. Step 5(b) identifies points where the number of choices exceeds the memory constraint, and splits the grammar at those points. Steps 5(c)-(h) handles or-grammars. In the case of terminals, they have to be accumulated into sets, and this is done in step 5(e) while ensuring that the size of any set does not exceed \( M \). Step 5(f)-(h) work on exposing non-terminals, if its memory requirement exceeds \( M \).

3. Implementation and Examples

In this section, we describe the implementation of the call-flow alteration system through grammar splitting. We describe the different types of grammars and show the splitting mechanism for each type of grammar. We use a sample doctor appointment call-flow to illustrate the effects of grammar splitting on the structure of the call-flow.

The grammar splitting is implemented in Java 2 (v1.5.0) and the generated corresponding altered dialogs were tested on the IBM WebSphere Voice toolkit that uses the IBM WebSphere Voice Server for speech recognition and speech synthesis. We tested the system for grammars of several working VXML dialogs and note that it generates syntactically correct dialogs. The implementation works over the SRGS-XML grammar format, however G-split is independent of the format.

3.1. Types of grammars

For grammars in the SRGS-XML format, we have used the Java API for XML Processing (JAXP) to build a parser that reads the grammars into a DOM tree. The number of terminals in the DOM tree provide the number of choices of that grammar. This number provides the memory that would be required to process the grammar. In case this memory requirement is greater than the device memory, the grammar has to be split for the call-flow to be accommodated in the device.

An SRGS grammar has a root node which specifies the structure of its children. As was explained in Section 2 grammars are of two types: and-grammar and an or-grammar. The and-grammar has the following structure for a root node:

\[
<\text{rule id="rootNode" scope="public">}
\text{<item>}
\text{<ruleref uri="#andPart1"/>}
\text{<ruleref uri="#andPart2"/>}
\text{<ruleref uri="#andPart3"/>}
\text{</item>}
\text{</rule>}
\]

There are three rule references defined within the \(<\text{item}>\) tag. These are the three required items in the grammar. Since all the three items are required, the grammar becomes an and-grammar. However, for a or-grammar, the root node has the alternative to choose one of the specified rule references. This is represented by the \(<\text{one-of}>\) tag in the grammar as shown below:

\[
<\text{rule id="rootNode" scope="public">}
\text{<one-of>}
\text{<item>ruleref uri="#forPart1"/></item>}
\text{<item>ruleref uri="#forPart2"/></item>}
\text{<item>ruleref uri="#forPart3"/></item>}
\text{</one-of>}
\text{</rule>}
\]

The or-grammar shown above expects only one of the items to be said by the user. So the parser understands the grammar structure (and type) by the use of \(<\text{one-of}>\) and \(<\text{item}>\) tags in the grammar. The parent of a leaf node in the grammar specifies the terminals of the grammar. Following is an example of terminals in the grammar:

To split an or-grammar, each of the rule references within a \(<\text{item}>\) tag can be moved to a separate grammar. However, depending on the memory that a device can support, some of the rule references can be combined in a single grammar. For splitting an and-grammar, rule references within the \(<\text{item}>\) tag and their corresponding definitions are put in a separate grammar. As was the case with or-grammars, the number of rule references that are contained in a particular split grammar would depend on the device memory. In case of an or-grammar, if the number of choices of a rule reference is too high, then this grammar is split into several split grammars, each having some number of choices from the parent grammar.

It is interesting to observe the effect of a grammar split on the structure of the call-flow. Split of an or-grammar at a particular node in the call-flow structure results in formation of a tree. The number of children of this node will be equal to the number of grammars in which the original grammar was split into. On the other hand, the split of a node having an and-grammar will result in a sequential structure at the node. The sequence will however be stretched by the number of grammars into which the original grammar would have been split.

3.2. Effect of splitting on call-flows

We illustrate the splitting mechanism for a doctor appointment call-flow. The call-flow is described by Figure 3.

```
<rule id="leafNode" scope="public">
  <one-of>
    <item>Choice A</item>
    <item>Choice B</item>
    <item>Choice C</item>
  </one-of>
</rule>
```

To split an or-grammar, each of the rule references within a \(<\text{item}>\) tag can be moved to a separate grammar. However, depending on the memory that a device can support, some of the rule references can be combined in a single grammar. For splitting an and-grammar, rule references within the \(<\text{item}>\) tag and their corresponding definitions are put in a separate grammar. As was the case with or-grammars, the number of rule references that are contained in a particular split grammar would depend on the device memory. In case of an or-grammar, if the number of choices of a rule reference is too high, then this grammar is split into several split grammars, each having some number of choices from the parent grammar.

It is interesting to observe the effect of a grammar split on the structure of the call-flow. Split of an or-grammar at a particular node in the call-flow structure results in formation of a tree. The number of children of this node will be equal to the number of grammars in which the original grammar was split into. On the other hand, the split of a node having an and-grammar will result in a sequential structure at the node. The sequence will however be stretched by the number of grammars into which the original grammar would have been split.

```
<rule id="leafNode" scope="public">
  <one-of>
    <item>Choice A</item>
    <item>Choice B</item>
    <item>Choice C</item>
  </one-of>
</rule>
```

To split an or-grammar, each of the rule references within a \(<\text{item}>\) tag can be moved to a separate grammar. However, depending on the memory that a device can support, some of the rule references can be combined in a single grammar. For splitting an and-grammar, rule references within the \(<\text{item}>\) tag and their corresponding definitions are put in a separate grammar. As was the case with or-grammars, the number of rule references that are contained in a particular split grammar would depend on the device memory. In case of an or-grammar, if the number of choices of a rule reference is too high, then this grammar is split into several split grammars, each having some number of choices from the parent grammar.

It is interesting to observe the effect of a grammar split on the structure of the call-flow. Split of an or-grammar at a particular node in the call-flow structure results in formation of a tree. The number of children of this node will be equal to the number of grammars in which the original grammar was split into. On the other hand, the split of a node having an and-grammar will result in a sequential structure at the node. The sequence will however be stretched by the number of grammars into which the original grammar would have been split.

In this example, the grammar used to decode the Dateutterance is an and-grammar. It captures the month and the day of the month in a single grammar. If the device memory is such that it will not be able to support the grammar since it has a large number of choices (365), then the grammar is split. However even after splitting, the system needs to know both the values, the day and the month to schedule an appointment. So the original and-grammar Date is split into two separate grammars. The figure also shows that when an and-grammar is split, the call-flow sequence is further stretched by introduction of an additional node in the sequence.

For the example shown in Figure 3, the total number of doctors was shown to be 10. Suppose that the total number of doctors were actually 1000. Thus the memory requirement of this grammar would be 1000. In order to split this or-grammar,
the intermediate node of the type of doctor is used to split the grammar. So the user is first asked about the type of doctor for which an appointment is required. Once the users specifies the choice, then the name grammar for that particular type of doctors is used in the call-flow. This reduces the memory requirement of the call-flow from 1000 to the maximum number of doctors within a particular type. Moreover, it is to be noted that since the original grammar was an or-grammar, its split resulted in the introduction of a tree-structured call-flow, even though the initial call-flow was sequential in nature.

4. Conclusion

We addressed the problem of automatically altering a dialogue call-flow so that it can meet memory constraints imposed by a pervasive device. The essence of the method lies in splitting a grammar whose memory requirements are larger than the device can support. The reduction in size has to be traded-off with an increase in the number of prompts (questions) in the dialogue call-flow. We present algorithm G-split, which can split both and-grammars and or-grammars.

G-split has practical implications on automatic dialogue call-flow adaptation for pervasive devices. Based on a device profile (characteristics), a call-flow can be automatically adapted for the particular device. We do not address the problem of generating prompts automatically.

More generally, the idea of extending call-flow adaptation for systems that use language models (rather than small “enumerated” grammars) coupled with an NLU engine would be interesting. Building a mechanism for adaptation in the absence of grammar operations appears to be a very challenging problem.

As speech applications become available on more and more devices, various interesting usability issues are likely to surface. Meeting the user expectation without having to manually customise a conversation for every person on every device is a worthy goal for the speech research community.

5. References