Kenneth J. Turner. Specifying and Realising Interactive Voice Services. In Harmut Koenig, Monica Heiner and Adam Wolisz, editors, Proc. Formal Techniques for Networked and Distributed Systems (FORTE XVI), Lecture Notes in Computer Science 2767, 15-30, copyright Springer-Verlag, Berlin, September 2003.

Specifying and Realising Interactive Voice Services

Kenneth J. Turner

Computing Science and Mathematics, University of Stirling, Scotland FK9 4LA
Email kjt@cs.stir.ac.uk

Abstract. VoiceXML (Voice Extended Markup Language) has become a major force in interactive voice services. However current approaches to creating VoiceXML services are rather low-level. Graphical representations of VoiceXML are close to the textual form of the language, and do not give a high-level description of a service. CRESS (Chisel Representation Employing Systematic Specification) can be used to give a more abstract, language-independent view of interactive voice services. CRESS is automatically compiled into VoiceXML for implementation, and into LOTOS (Language Of Temporal Ordering Specification) or SDL (Specification and Description Language) for automated analysis. The paper explains how CRESS is translated into VoiceXML and LOTOS.

1 Introduction

1.1 Motivation

This paper explains how to represent, specify and analyse IVR (Interactive Voice Response) services. VoiceXML (Voice Extended Markup Language [13]) is typically used to implement automated telephone enquiry systems. VoiceXML is much more acceptable to users than the early generation of touch-tone systems. Specifically, VoiceXML allows users to do what they expect in a telephone call: talk and listen. VoiceXML can be linked to databases, telephone networks and web servers. As a result, VoiceXML is very useful for those who cannot directly access such information. A user on the move, for example, is likely to have a mobile telephone but limited web access. A partially sighted or physically handicapped user could find web-based services difficult or impossible to use. Many households still do not have web access.

Being an application of XML, VoiceXML is textual in form. However several commercial packages (e.g. Covigo Studio, Nuance V-Builder, Voxeo Designer) provide a graphical representation. Some of these reflect the hierarchical structure of VoiceXML, while others emphasise the relationship among VoiceXML elements. These packages are (not surprisingly) very close to VoiceXML and do not give a clear overview of interactive voice services. In the author's opinion, existing graphical formats are 'window dressing' that do little to clarify the structure and flow of VoiceXML scripts. It is easy, even common, to write VoiceXML scripts whose flow of control is obscure and hard to follow. Indeed, VoiceXML can suffer from the 'spaghetti code' (tangled logic) that structured programming was devised to avoid. VoiceXML adopts a pragmatic and programmatic approach. There is no way to formally check or analyse a VoiceXML script.

In telephony, services are often composed from self-contained features. A feature is an additional function that is triggered automatically (e.g. call forwarding or call

screening). Because a feature is triggered and not explicitly called, it readily adds supplementary capabilities. The value of features has been amply demonstrated in the IN (Intelligent Network). VoiceXML does not have features (though it has subdialogues). In fact, VoiceXML does not directly recognise the concept of a service. It is therefore be useful to enhance VoiceXML with mechanisms for services and features.

The author's approach to defining and analysing services is a graphical notation called CRESS (Chisel Representation Employing Systematic Specification). CRESS was initially based on the industrial Chisel notation developed by BellCore [1]. However, CRESS has considerably advanced from its beginnings. The aim of using CRESS with VoiceXML is to define key aspects of interactive voice services. The advantages of CRESS over using VoiceXML directly are:

– VoiceXML is very close to implementation. However CRESS services are represented at a more abstract level, making it easier to grasp their essence. For the same reason CRESS diagrams can be translated into a number of target languages, of which VoiceXML is just one.
– There is no formal definition of VoiceXML. Some concepts in VoiceXML are only vaguely described (e.g. event handling) and some are defined loosely (e.g. the semantics of expressions and variables). As a result, it is impossible to say for certain what certain VoiceXML constructs mean. At times the author has had to resort a commercial VoiceXML implementation to discover what some constructs might mean. Even then, the commercial solution has been seen to behave implausibly. Through translation to a formal language, CRESS contributes to a more precise understanding of VoiceXML.
– A large VoiceXML application typically has many documents with many parts. It can be difficult to check whether the application is self-consistent, e.g. will not loop indefinitely or end prematurely. VoiceXML development in practice uses manual debugging. CRESS gives the immediate benefit of translation to a formal language: LOTOS (Language of Temporal Ordering Specification) and SDL (Specification and Description Language). The resulting specification can be rigorously analysed.

1.2 Relationship to Other Work

Several graphical representations have been used to describe communications services. SDL is the main formal language used in communications. Although it has a graphical form, SDL is a general-purpose language that was not designed particularly to represent communications services. MSCs (Message Sequence Charts) are higher-level and more straightforward in their representation of services. UCMs (Use Case Maps [2]) have been used to describe communications services graphically. However none of these approaches is domain-specific, and they cannot be translated into a range of languages. In comparison to CRESS, SDL for example does not have specialised support for a domain like interactive voice services. As a result the equivalent SDL specification is larger and more complex. The only formal analysis possible is whatever SDL offers (mainly state space exploration). With CRESS an SDL-based analysis remains possible, different kinds of analysis can be achieved through LOTOS, and VoiceXML scripts can be obtained automatically *from the same diagrams*. See for example [4, 9] for a comparison of CRESS and SDL descriptions of SIP (Session Initiation Protocol).

As noted earlier, there are a number of commercial tools for VoiceXML. These offer more complete VoiceXML coverage than CRESS, and provide proprietary extensions for commercial deployment. However they are focused on VoiceXML *only*, and do not offer any kind of formal analysis. Their (graphical) representations are too close to VoiceXML for abstract service descriptions that are comprehensible to non-specialists.

Although CRESS has origins in communications services, it is not tied to these. CRESS has plug-in domains that define the service vocabulary in a separate and modular fashion. CRESS has already been proven with services for the IN (Intelligent Network) [8] and SIP (Session Initiation Protocol) [9, 11]. The work reported in the present paper shows how CRESS can be used with VoiceXML.

CRESS is a front-end for defining and formalising services. CRESS is neutral with respect to the target language. The translation of CRESS into LOTOS or SDL gives formal meaning to services defined in CRESS. This formalisation provides access to any analytic technique using these languages. Among these, the author's own approach [7] is one of several. For implementation, CRESS can also be compiled as appropriate into SIP CGI (Common Gateway Interface, realised in Perl), partly into SIP CPL (Call Processing Language), and also into VoiceXML.

A key issue in telephony is feature interaction [3] – independently designed features can interfere with each other. This issue is well known from traditional telephony and the IN, but also arises with SIP services. The feature interaction literature is too large to review here; see, for example, the proceedings of FIW (Feature Interaction Workshop). Although VoiceXML does not recognise the concept of service or feature, it has been shown that feature interactions can also arise with VoiceXML [11].

1.3 Overview of The Paper

The new contribution made by this paper is the application of CRESS to IVR services. The paper discusses how IVR services and features can be described in CRESS, and explains how they are translated into VoiceXML and LOTOS. CRESS can also be translated into SDL, as outlined in [10]. As background, section 2 summarises the CRESS graphical notation insofar as it applies to interactive voice services. Examples of CRESS diagrams appear later, and CRESS is further discussed in [8, 9, 11]. Section 3 introduces VoiceXML and its representation in CRESS. It will be seen how CRESS diagrams for interactive voice services are translated into VoiceXML. Section 4 discusses how the same diagrams are translated into LOTOS. This allows a variety of formal analyses to be carried out of a service before it is developed and deployed using VoiceXML.

2 The CRESS Notation

At first sight, it might seem that CRESS is just another way of drawing state diagrams. However it differs in a number of important respects. State is intentionally implicit in CRESS because this allows more abstract descriptions to be given. Arcs between states may be guarded by event conditions as well as value conditions. Perhaps most importantly, CRESS has explicit support for defining and composing features. CRESS also has plug-in vocabularies that adapt it for different application domains. These allow CRESS diagrams to be thoroughly checked for syntactic and static semantic correctness.

2.1 Diagram Elements

Ultimately, CRESS deals with a single diagram. However it is convenient to construct diagrams from smaller pieces. A multi-page diagram, for example, is linked through connectors. More usefully, features are defined in separate diagrams that are automatically included by either cut-and-paste or by triggering. A CRESS diagram is a directed, possibly cyclic graph. If the graph is cyclic, it may not be possible to determine the initial node uniquely. In such a case, an explicit **Start** node is given. Comments may take several forms: text between parallel lines, hyperlinks to files, and audio commentary.

Nodes in a diagram (shown as ovals) contain events and their parameters (e.g. **Submit** *order.jsp ″weight product″*). A node is identified by a number followed optionally by a symbol to indicate its kind. For example, the first node of a template feature is marked '+' if it is appended to the triggering node, or '–' if it is prefixed. Events may be signals (input or output messages) or actions (like programming language statements). A **NoEvent** (or empty) node can be used to connect other nodes. An event may be followed by assignments separated by '/' (e.g. */ timeout* <– *2*).

The arcs between nodes may be labelled by guards. These may be either value conditions (imposing a restriction on the behaviour) or event conditions (that are activated by dynamic occurrence of an event). Event handlers are distinguished by their names (e.g. **NoInput**, triggered when the user does not respond to a VoiceXML prompt).

A CRESS diagram may contain a rule box (a rounded rectangle) that defines general rules and configuration information. A rule box typically declares the types of diagram variables (e.g. **Uses Value** *product, weight*). A rule box may define configuration information like parent diagrams, chosen features and translator options. Rule boxes have yet other uses [8, 9, 11] that are not so applicable to interactive voice services.

The main CRESS diagram defines the root behaviour. Although this may be the only diagram, CRESS supports feature diagrams that modify the root diagram (or other features). A spliced (plug-in) feature is inserted into a root diagram by cut-and-paste. The feature indicates how it is linked into the original diagram by giving the insertion point and how it flows back into the root diagram. This style of feature is appropriate for a one-off change to the original diagram.

It is often preferable to use a template (macro) feature that is triggered by some event in the root diagram. The triggering event is given in the first node of the feature. Feature execution stops on reaching a **Finish** (or empty) node. At this point, behaviour resumes from the triggering node in the original diagram. A template feature is statically instantiated using the parameters of the triggering event. The instantiated feature may be appended, prefixed or substituted for the triggering node.

2.2 Automated Support

CRESS usually relies on a domain-specific infrastructure. For example, IVR services often require a speech synthesiser and a speech recogniser that cooperate with the main application. Such a framework is specified using the same target language as the one to which diagrams are compiled (e.g. LOTOS, SDL, VoiceXML). Although the framework is specific to a domain and a target language, it is independent of the particular services

or features deployed. The framework includes macro calls that activate the CRESS preprocessor. This automatically generates configuration-specific information such as the translated diagrams and feature-dependent data types.

The CRESS toolset has the form of a conventional compiler but is unusual in some respects. For portability it is written in Perl, comprising about 13,000 lines of code. The CRESS toolset consists of five main tools. Including test scenarios, there are about 600 supporting files for all domains and target languages. Internally the CRESS toolset consists of a preprocessor (that instantiates the specification framework), a lexical analyser (that deals with various diagram formats), a parser (that performs syntactic analysis), and several code generators (e.g. for LOTOS, SDL and VoiceXML). Although it might have been desirable to use a parser generator (e.g. Antlr), parsing is only a small part of what CRESS has to do. A traditional compiler deals with textual languages. CRESS, however, deals with a graphical language. This creates interesting challenges, e.g. compiling cyclic rather than hierarchical constructs.

3 Interactive Voice Services in VoiceXML

3.1 Introduction to VoiceXML

VoiceXML [13] derives from earlier work on scripting languages for interactive voice services. VoiceXML is a mixture of the declarative and the imperative, the event-driven and the sequential. VoiceXML is a large language embedded in an even larger framework. For example, VoiceXML includes ECMASCRIPT (JavaScript). It also supports complex grammars for speech synthesis and speech recognition. VoiceXML is integrated with other technologies such as database access and web servers.

VoiceXML lacks the telephony concept of a feature as behaviour that may be triggered by some condition. The nearest equivalent in VoiceXML is a subdialogue (resembling a subroutine). Subdialogues are executed in an independent interpreter context, making it difficult to share certain information. In VoiceXML, at best some code can be *explicitly* invoked as a 'feature'; automatic feature invocation is not supported. Triggered features have proven their worth in telephony and are supported by CRESS.

The VoiceXML caller completes fields in forms (or menus) by speaking in response to prompts. Each field is associated with a variable that is set to the user's input. VoiceXML applications are often written as a number of documents containing a number of forms, each containing a number of fields. This is the most natural form of modularity in VoiceXML. However this can easily hide the flow between the forms and fields.

TTS (Text-To-Speech) may be used to speak messages. Text may be marked up to indicate how certain words are pronounced, and generally to define the speech pattern. However TTS is only approximation of natural speech, so VoiceXML allows prerecorded voice to be used in preference.

Speech recognition is used to extract digital data from user input. This is guided by a speech grammar, for which there are several standards. Numeric inputs (including menu choices and yes/no) can also be provided using DTMF (Dual Tone Multi-Frequency), i.e. dialling digits on a touch-tone telephone. CRESS supports the standard VoiceXML grammars: *Boolean* (optionally defining DTMF digits for yes/no), *Currency*, *Date*,

Digits (optionally defining expected length, or minimum/maximum lengths), *Number*, *Phone* (with optional extension) and *Time* (12/24 hour clock).

Some VoiceXML actions may be governed by a count or a condition on when the action is permitted. For example a different prompt may be given on the third input attempt, or a field may be selected only when some condition is true. Actions may have optional parameters (e.g. a sound file or fetch timeout) that are relevant to a VoiceXML platform but not directly useful for CRESS. Although these may appear in a CRESS diagram, they are used only when translating into VoiceXML. For other target languages these optional parameters are ignored.

The types supported by CRESS are domain-specific. For VoiceXML there is just a single type, *Value*, since the underlying ECMASCRIPT has dynamic types. Actual values can be booleans, numbers or strings. In addition there are *null* and *undefined* values.

VoiceXML supports a fairly complex hierarchical event model. Event handlers may be defined at four levels: platform, application, form, field. Platform handlers provide fall-back support, though they are rarely useful. Application handlers oversee all forms in an application. Form handlers allow their fields to share common event handling. Finally, fields usually define handlers for events of specific interest. A script may <**throw**> an event, transferring control a matching handler. Standard events include:

Cancel, Exit, Help: the user asked to cancel processing, to exit, or to get guidance

Catch: deals with a list of events

Error: a run-time error occurred

Filled, NoInput, NoMatch: the user spoke correct input, nothing, or invalid input.

Although VoiceXML does not consider **Filled** to be an event, it behaves like one. Besides standard events, programmer-defined events may be constructed from several parts (e.g. *login.failure.password*). Normally this would be caught by a handler for the same name. But if there is nothing to match, a handler for *login.failure* (or failing that *login*) may deal with the event. If no handler matches, the application terminates.

Events are also implicitly associated with a prompt count. Each time a field is entered, its prompt count is incremented. This may be used vary the response to an event. In fact this is more complex than it seems. Suppose event handlers are defined for counts 1 (the default), 2 and 4. The first is activated on count 1, the second on counts 2 or 3, and the last on count 4 or higher. A condition may also be imposed on an event handler being activated. This is relevant if several handlers could otherwise apply. VoiceXML does not define what happens if conditions overlap – in fact the behaviour is non-deterministic.

3.2 CRESS for VoiceXML

In principle, VoiceXML has elements at platform, application, form and field levels. VoiceXML can also be split over a number of documents. However a VoiceXML application can be defined as a single document with a single form, and this is how it is regarded in CRESS; in fact, application and form level are the same in CRESS. Fields can be defined as separate sections or pages of a CRESS diagram, using connectors to join them. For a large application this is convenient and more modular. However for a small application it is sufficient to represent the form as a single integrated whole. In

addition, splitting fields makes the flow between them less obvious. For these reasons, fields are deliberately not prominent in CRESS.

CRESS is not a literal graphical rendering of VoiceXML structure. This would be pointless since most commercial tools for VoiceXML do this anyway. The flow of control in CRESS can be more visible; in VoiceXML it can be hard to determine. The flow is sometimes implicit (e.g. transitioning to the next field on completion of the current one) and sometimes buried (e.g. an embedded <**goto**>). CRESS supports cyclic behaviour as loops in a diagram; these have to be coded indirectly in VoiceXML.

CRESS expects to have a definition of root behaviour as the core of a service. In VoiceXML, an application root document serves a similar purpose but is very restrictive. It may contain only variables, event handlers and elementary definitions that are common to the documents of a VoiceXML application.

It is not feasible for CRESS to support the entirety of VoiceXML, ECMASCRIPT, speech synthesis markup, speech grammars, database access and web access. Instead, CRESS concentrates on the essential aspects of VoiceXML control. Limited support is provided for ECMASCRIPT – specifically for numerical, string and logical expressions.

The following summarises the main elements of CRESS for interactive voice services. Unless stated, the VoiceXML equivalent is very similar (e.g. **Audio** in CRESS corresponds to <**audio**> in VoiceXML). Strings and lists are given in double quotes (e.g. ″Please place your order″, ″weight product″). Substrings can be given inside a list using single quotes. Variables, grammars, etc. are without quotes in CRESS. VoiceXML actions sometimes allow literals or expressions as alternative parameters (e.g. a literal or computed event name may be thrown); CRESS always uses an expression.

Audio *message* speaks a message. Variable values in this or any text string may be interpolated in CRESS, using *$variable* to include the value of a variable. Some names are special, e.g. *$enumerate* is used to include the options of the current field. The VoiceXML equivalents of these are <**value**> and <**enumerate**> respectively.

Clear *[variables]* resets the prompt count, and undefines all (or the named) variables.

Exit leaves the application.

Option *variable prompt options [condition]* defines a <**field**>, issues a <**prompt**>, analyses the input using <**option**> values, and sets the field variable from this. An optional condition can be imposed on entry to the field.

Prompt *message [count] [condition]* issues an audio prompt. The prompt may be subject to a count and/or a condition.

Reprompt restarts the current form. The first unfilled field is entered, usually causing the most recent prompt to be re-issued.

Request *variable prompt grammar [condition]* behaves like **Option**, but defines acceptable input using a grammar rather than a list of specific options.

Retry restarts the current form, re-inputting the most recent field. This is a common requirement that is surprisingly missing from VoiceXML. It undefines the current field variable, and issues a <**reprompt**>.

Subdialog[ue] *URI variables* send the variable values to a URI (either another VoiceXML script or a web server executable script). In general, this may return a new VoiceXML script dynamically. This is not a problem when CRESS is interpreted

by VoiceXML. However for translation to other languages (e.g. LOTOS or SDL), it is not possible to handle dynamically created VoiceXML. Instead, limited support is provided for external scripts that perform a computation (e.g. interrogate a database) and return results. CRESS provides a web adaptor written in C that links to the target language.

Submit *URI variables* sends the variable values to a web server URI (usually an executable script). Often the server just absorbs the results (e.g. writes them to a database). As for subdialogues, the server may return VoiceXML created on-the-fly. This cannot be handled except when VoiceXML is the target language. The CRESS approach handles the commonest cases of server scripts that produce no result (**Submit**) and scripts that compute some results (**Subdialogue**).

Throw *event* passes an event to the closest matching handler.

3.3 Sample Interactive Voice Services

To illustrate the notation, Figure 1 shows the CRESS root diagram for a VoiceXML application. This is for the hypothetical Charities Bank that allows telephone donations to charity. This service invites the caller to name the *charity* (UNICEF, WWF, Red Cross) and the required *amount* in US dollars. These items are then submitted to the *donation.jsp* script. If the user asks for help or says nothing following a prompt, an explanation is given and the user is re-prompted. VoiceXML reads a currency amount as a string whose first three characters give the currency code (e.g. ″USD″). In case the user says another currency (e.g. pounds sterling), the user is re-prompted if the stated currency is not US dollars. **Retry** in node 7 is used to clear the value entered for *amount*, otherwise the field will be ignored on the re-prompt because it has already been filled.

Suppose that Charities Bank has a range of applications besides the donation application in Figure 1. There might, for example, be separate applications to enquire about what charities are supported, or to request a tax relief statement. It would be desirable to ensure a consistent VoiceXML treatment of all these applications: there should be the same default handling of events and a common introduction. It would also be worthwhile to request confirmation before anything is submitted to a web server. There is therefore a case for common features. For brevity several features are omitted here, such as ones to request an account number and a PIN.

Figure 2 is a feature that defines an introductory environment for all bank applications. The feature is placed just after the **Start** node in the root diagram (implicit before Figure 1 node 1). Welcome messages are spoken before executing application-specific code. Common handlers are defined for various events. Although an application is likely to deal with **NoInput** and **NoMatch** on a per-field basis, figure 2 ensures that after three such failures the user is disconnected. Figure 2 also defines general VoiceXML properties: here the timeout for no input is set to two seconds (*timeout* $<-$ *2*).

Figure 3 defines a confirmation feature that asks the user to proceed before submitting information to a web server. This feature is not specific to Charities Bank, and is usable in a number of applications. The feature is triggered by a **Submit** action, being executed just before it. On user confirmation, execution continues with submission; otherwise, the current fields are cleared and the user is re-prompted.

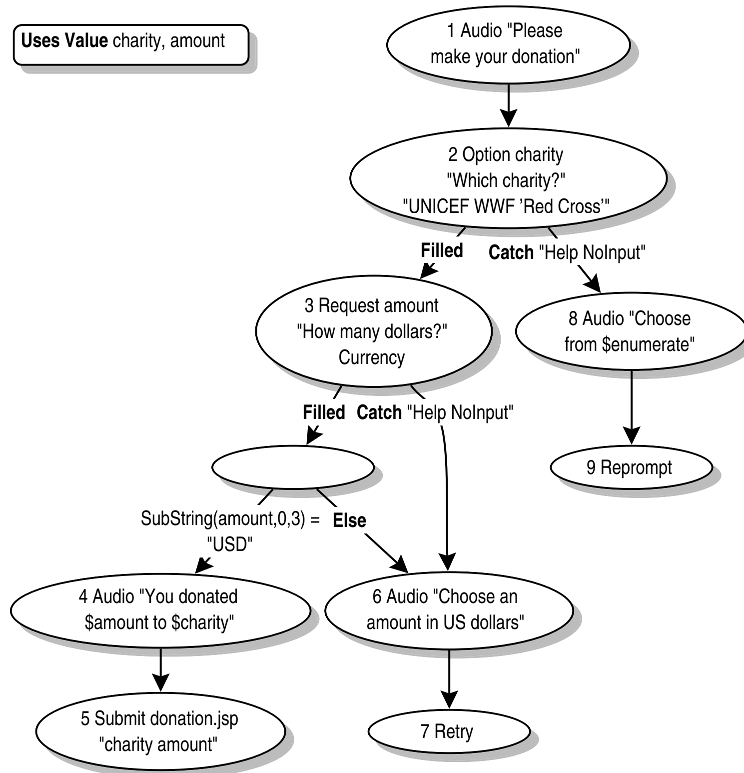

Fig. 1. CRESS Root Diagram (*Donate*) for Charity Donation Application

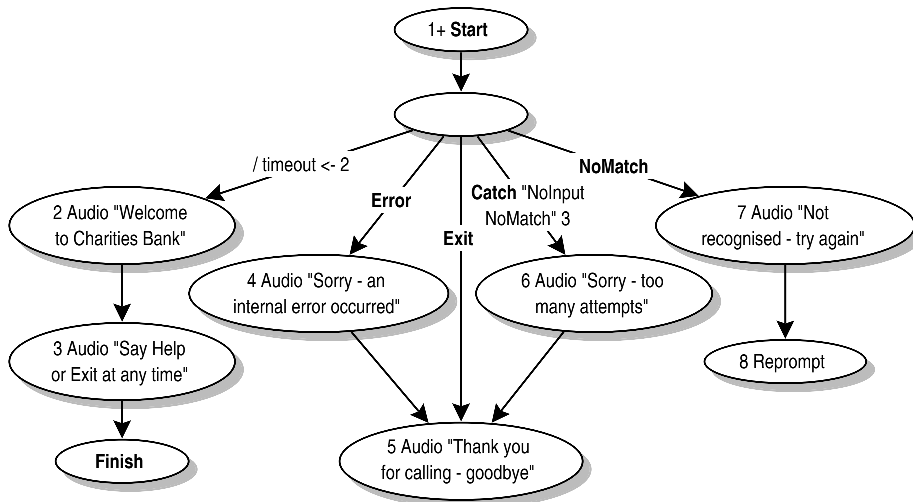

Fig. 2. CRESS Feature Diagram (*Intro*) to introduce Charities Bank Applications

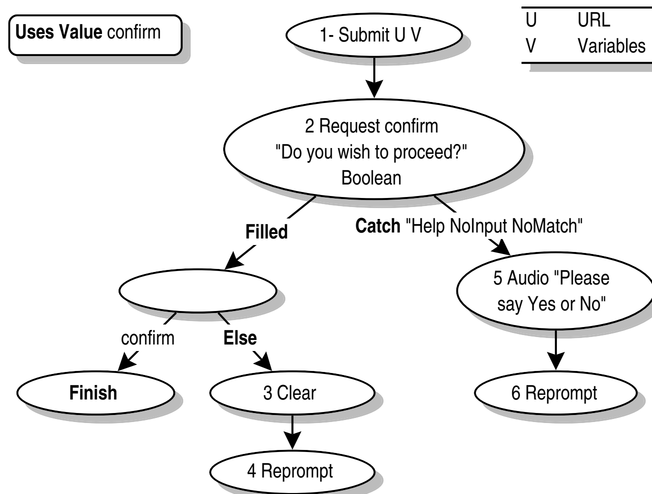

Fig. 3. CRESS Feature Diagram (*Confirm*) for Confirmation

Most of the translation from CRESS to VoiceXML is straightforward. One complication that arises is how to deal with loops in diagrams and nodes that can be reached along more than one path. It might seem obvious to use a VoiceXML <**goto**>. Unfortunately this may branch only to a document, form or field; it is not possible to move to an arbitrary node. As a result, it is necessary to branch using an event. (A <**throw**> acts as a computed <**goto**> anyway.) The revisited node is then translated as an event handler. Most CRESS operators have equivalents in VoiceXML, but a few like *After* (remove a prefix from a string) and *In* (set membership) are defined using ECMASCRIPT.

To give an idea of how CRESS translates interactive voice services into VoiceXML, the following shows some of the translation for figure 1 as modified by the features in figures 2 and 3. As shown, the CRESS translator automatically generates extensive comments that link the code to the diagrams. The CRESS **Request** in figure 1 node 3 becomes a VoiceXML field that fills in the *amount* as a currency. The user is prompted to enter a donation in dollars. If the user asks for help or does not say anything, an event handler catches this and moves to figure 1 node 6. If US dollars are specified, the donation is announced to the user. Execution then continues with the *confirm* field; *Confirm.1* is instance 1 of the *Confirm* template.

```
<field name='amount' type='currency'>                      <!-- Donate 3 field 'amount' -->
  <prompt>How many dollars?</prompt>                               <!-- Donate 3 prompt -->
  <catch event='help noinput'>                                          <!-- catch event -->
    <throw event='donation.6'/>                                 <!-- Donate 6 (again) -->
  </catch>                                                                <!-- end catch -->
  <filled>                                                             <!-- filled event -->
    <if cond='amount.substring(0,3) == "USD"'>          <!-- check SubString = "USD" -->
      <audio>                                                       <!-- Donate 4 audio -->
        You donated <value expr='amount'/> to <value expr='charity'/>
      </audio>
```

```
    <goto nextitem='confirm.1'/>                       <!-- to Confirm.1 2 -->
  <else/>                                  <!-- else after SubString = "USD" -->
    <throw event='donation.6'/>                     <!-- Donate 6 (again) -->
  </if>                                    <!-- end check SubString = "USD" -->
 </filled>                                               <!-- end filled -->
</field>                                                  <!-- end field -->
```

As noted earlier, figure 1 node 6 must be translated in an event handler because there are two paths to it. It provides audio help to the user. The CRESS **Retry** undefines the *amount* field (if necessary, forcing re-entry to the field) and re-prompts the user.

```
<catch event='donation.6'>                                 <!-- Donate 6 -->
  <audio>Choose an amount in US dollars</audio>      <!-- Donate 6 audio -->
  <assign name='amount' expr='undefined'/>  <!-- Donate 7 undefine 'amount' -->
  <reprompt/>                                   <!-- Donate 7 to form top -->
</catch>                                                   <!-- end catch -->
```

4 Interactive Voice Services in LOTOS

In comparison to the translation of CRESS diagrams into VoiceXML, the translation into LOTOS is extremely tricky. In addition, a very substantial specification framework is required. This is fixed and predefined, being completed with types and behaviour specific to the diagrams being translated. The complexity of the translation should be no surprise since much of a VoiceXML interpreter has to be represented in LOTOS.

4.1 Inputs, Outputs and Actions

Normally each node is translated directly into LOTOS behaviour. However if there is more than one path to a node, this node and the following ones are translated as a LOTOS process. The branches to the node then become calls of this process. Since an event handler may be entered repeatedly, a node following an event guard is also translated as a process.

The CRESS parser optimises diagrams before they are passed to a code generator. For example **NoEvent** nodes are removed where possible, and **Else** branches are moved to the end of the guard list. However it is not possible to remove a **NoEvent** node in some circumstances (e.g. in a loop or between guards, see figure 1 before nodes 4 and 6). A **NoEvent** may thus not need translation or may be translated as a process.

Inputs and outputs are reasonably straightforward to translate. It is necessary to distinguish them because inputs may accept new values, while outputs must use only defined values. In fact the CRESS translator performs a data-flow analysis to determine this. If an input variable is known at a certain point, it is preceded by '!' in the LOTOS translation; if an input variable is unknown, it is preceded by '?'.

CRESS nodes may also be VoiceXML actions that do not simply input or output and so are classed separately. Actions are domain-specific, so their translation into LOTOS also depends on the domain. For most actions, the translation is not too complex. The main exceptions are fields (**Menu**, **Option**, **Request**) and events (**Throw**).

Each field is translated to a LOTOS process. If a VoiceXML field has already been filled in (its field variable is defined), behaviour continues with the next field. CRESS

must therefore statically build a map of which field follows which. Since VoiceXML is linear but CRESS diagrams are two-dimensional, it may not be clear what the next field is. By convention, fields are ordered by node number within each diagram. A field is entered if its field variable is undefined and its condition is satisfied. The prompt count is incremented at this point.

Input recognition is performed by a predefined *Recogniser* process that deals with all the standard VoiceXML grammars. This is much simpler than speech recognition, but is still complex (in a language like LOTOS). The LOTOS recogniser does not have to handle the variations that occur in speech. For example a currency amount might be spoken as 'one hundred and ten dollars' or 'a hundred ten bucks'. The LOTOS recogniser also accepts the DTMF equivalent of 'speech' input. The recogniser deals with events like **Cancel**, **Exit**, **Help** and **NoInput**. Once recognition is complete, the recogniser synchronises with the application on the resulting event. For **Filled**, the input value is also supplied. The field process then throws the event it received, causing behaviour to continue with the corresponding event handler. Although this might seem a complex solution, it is exactly what VoiceXML does. It is also essential because the same event may be handled differently according to the current prompt count and condition. For example, **NoInput** may be handled at field level (figure 1 nodes 6 and 8) or at form level (figure 2 node 6).

4.2 Expressions and Expression Guards

Interactive voice services expect to use speech synthesis and speech recognition. It is not, of course, meaningful to use speech with LOTOS. Instead, speech is simulated using text messages. Synthesis is little more than string processing, but recognition requires substantial machinery for parsing text input according to the standard grammars. The LOTOS specification framework includes about 900 lines of intricate data types. These are complex partly because LOTOS lacks built-in support for types like characters, numbers and strings. However, the recogniser also requires VoiceXML-specific data types.

ECMASCRIPT numerical, string and logical operators are supported by equivalent LOTOS operators. The dynamic types of VoiceXML create a problem for translation since LOTOS is strongly typed. All variables and values are therefore translated to a single *Value* type in LOTOS that is interpreted according to the specific value. Assignment is made using a LOTOS **Let** statement. As well as the declared diagram variables, there are two implicit ones: *vxoptions* (the current <**option**> values) and *vxprompt* (the current prompt counter). All these variables are parameters of the generated processes.

Expression guards are straightforward to translate. The only complication is that a VoiceXML *Value* must be translated to a LOTOS *Bool*. The convenience syntax **Else** in CRESS is handled by accumulating all other expression guards and negating them. It is possible to give only specific guard expressions without **Else**. In this case, a translator option can be set to deal with guards that leave gaps (e.g. '$n > 0$' and '$n < 0$').

4.3 Events and Event Guards

Event handling is very complex to translate. As explained in section 3.1, events may be handled at multiple levels, using multi-part event names, subject to a prompt count and

a boolean condition. In addition, a VoiceXML platform provides default handlers for all standard events. In the generated LOTOS, platform handlers are defined for these in processes *Event_1*, etc.

The generated LOTOS also defines process *Event_0* as the main event dispatcher, called whenever an event is thrown. The CRESS translator statically builds a table of contexts and events. A context value is either 0 (i.e. application/form level) or > 0 (meaning a field number). All the events that may occur in a context are extracted from the CRESS description. It would be simplest if the destination of a thrown event name could be determined statically. Unfortunately this is not possible because the thrown event can be computed dynamically as an expression (e.g. *"login.failure."* + *cause*). The CRESS translator generates event dispatcher code that respects the priority of VoiceXML event handling: longer event prefixes and higher prompt counts take precedence.

To give an idea of how this is done, here is an extract from the event dispatcher for field 2 (figure 1 node 3). For brevity, process gates and parameters are omitted below. *Donate* is the main application (figure 1), *Intro_1* is the first instantiation of the introduction feature (figure 2). Node numbers are appended to these labels, with *None* and a count for a **NoInput** node.

```
[field Eq 2]->                                                        (* field Donate 3? *)
  (
    [Match(event,Cancel)]-> Event_1                                   (* Cancel? *)
  []
    [Match(event,Error)]-> Intro_1_4                                  (* Error? *)
  []
    [Match(event,Filled)]-> Donate_None_0                             (* Filled? *)
  []
    [Match(event,Help)]-> Donate_6                                    (* Help? *)
  []
    [Match(event,NoInput) And (vxprompt Ge 3)]-> Intro_1_6            (* NoInput 3? *)
  []
    [Not(Match(event,NoInput) And (vxprompt Ge 3))]->                 (* Else *)
      (
        [Match(event,NoInput)]-> Donate_6                             (* NoInput? *)
      []
        [Match(event,NoMatch) And (vxprompt Ge 3)]-> Intro_1_6        (* NoMatch 3? *)
      []
        [Not(Match(event,NoMatch) And (vxprompt Ge 3))]->             (* Else *)
          (
            [Match(event,NoMatch)]-> Intro_1_7                        (* NoMatch? *)
          []
            [Match(event,Xit)]-> Intro_1_5                            (* Exit? *)
          )
      )
  )
```

4.4 CRESS Translation to LOTOS

To give an idea of how CRESS translates interactive voice services into LOTOS, the code below was generated for the VoiceXML example in section 3.3.

The following is an extract from the body of process *Donate_3* (figure 1 node 3). Behaviour continues to the next field (*Confirm* node 2) if the field variable (*amount*) is undefined. Otherwise the options list is emptied (node 3 defines a **Request** not an **Option**), and the prompt count is incremented. The prompt and grammar are sent to the recogniser. Its response synchronises with the VoiceXML application, causing an event to be handled by the event dispatcher (*Event_0*). The prompt count is then reset. For readability, string values are shown below in conventional form, though LOTOS requires an awkward syntax using '+' for character concatenation.

```
 [(amount Ne Undefined) Of Bool]->                                 (* ignore field? *)
   Confirm_1_2                                                     (* to Confirm.1 2 *)
[]
 [(amount Eq Undefined) Of Bool]->                                 (* enter field? *)
   (
     Let vxoptions:Text = <>, vxprompt:Nat = vxprompt + 1 In       (* update locals *)
       Recogniser !request !"How many dollars?" !Currency Of Grammar;(* request field *)
       (
         Recogniser !Filled ?amount:Value;                         (* filled event *)
         (
           Let vxprompt:Nat = 0 In                                 (* reset prompt count *)
             Event_0                                               (* dispatch event *)
         )
       []
         Recogniser ?event:Event;                                  (* other event *)
         Event_0                                                   (* dispatch event *)
       )
   )
```

The following is an extract from the body of process *Donate_6* (figure 1 node 6). The field variable (*amount*) is undefined, and field processing restarts from the top of the form (process *Donate_2*).

```
User !Audio !"Choose an amount in dollars";                        (* Donate 6 *)
(
 Let amount:Value = Undefined In                                   (* update local *)
   Donate_2                                                        (* Donate retry 7 *)
)
```

Once a translation to LOTOS has been obtained, the interesting work can begin. The author has used both TOPO and CADP with the resulting LOTOS. The specification can be simulated, though this is not very useful since CRESS might as well be translated to VoiceXML and executed as normal. Where the LOTOS translation comes into its own is the formal analysis. Below are examples of what LOTOS can be used for, but VoiceXML cannot. Although there is insufficient space here to say more, the referenced papers can be consulted for additional detail.

– The specification can be formally analysed to detect deadlocks, livelocks, unreachable states and unspecified receptions. For example, it is easy to write VoiceXML scripts that loop indefinitely. Figure 1 suffers from this; the reader is challenged to detect the problem! Without extensive and time-consuming testing, this can be hard to find with VoiceXML.

– Tests can be automatically generated from the specification. For example, the author has developed PCL (Parameter Constraint Language [12]) to allow practical tests to be created from data-intensive specifications (such as interactive voice services). If the specification has finite behaviour, exhaustive tests can be generated that traverse all paths. If the specification has infinite behaviour, tests must be generated as Chinese Postman tours of the specification's suspension automaton [6]. The tests form a useful (regression) test suite for live testing of an IVR service. In this context, the tests can act as scripts for human users or can be fed into a speech synthesiser acting as an automated caller.
– Desirable properties of the service can be formulated in ACTL or XTL, e.g. as done in [5]. For example these might include 'a bank account must not be debited without the correct PIN' (safety), 'a call must end with a goodbye message' (liveness), or 'the same prompt must not be issued more than three times' (freedom from loops). The CADP model checker can verify such properties against the generated LOTOS.
– Feature interactions can be checked [11]. For example, a feature that introduces extra choices in a menu can interfere with current use of DTMF digits to select from the menu. A feature may introduce an event handler that overrides the form event handler, resulting in different behaviour. An interaction can also arise if two features change a variable in inconsistent ways.

5 Conclusion

It has been shown that CRESS can represent interactive voice services. It has been seen how CRESS descriptions can be translated into VoiceXML and into LOTOS. CRESS combines the benefits of an accessible graphical notation, automated implementation of a VoiceXML application, and formal analysis of problems in a service description. CRESS is thus valuable as an aid to developing interactive voice services.

The plug-in architecture of CRESS has now been demonstrated in three different domains: conventional telephony (IN), Internet telephony (SIP), and interactive voice (VoiceXML). Although these are all examples of voice services, the approach is generic and should be relevant to non-voice applications such as web services. For example, it is hoped in future to apply CRESS to WSDL (Web Services Description Language).

References

1. A. V. Aho, S. Gallagher, N. D. Griffeth, C. R. Schell, and D. F. Swayne. SCF3/Sculptor with Chisel: Requirements engineering for communications services. In K. Kimbler and W. Bouma, editors, *Proc. 5th. Feature Interactions in Telecommunications and Software Systems*, pages 45–63. IOS Press, Amsterdam, Netherlands, Sept. 1998.
2. D. Amyot, L. Charfi, N. Gorse, T. Gray, L. M. S. Logrippo, J. Sincennes, B. Stepien, and T. Ware. Feature description and feature interaction analysis with use case maps and LOTOS. In M. H. Calder and E. H. Magill, editors, *Proc. 6th. Feature Interactions in Telecommunications and Software Systems*, pages 274–289, Amsterdam, Netherlands, May 2000. IOS Press.

3. E. J. Cameron, N. D. Griffeth, Y.-J. Lin, M. E. Nilson, W. K. Schnure, and H. Velthuijsen. A feature-interaction benchmark for IN and beyond. *IEEE Communications Magazine*, pages 64–69, Mar. 1993.
4. K. Y. Chan and G. von Bochmann. Methods for designing SIP services in SDL with fewer feature interactions. In D. Amyot and L. Logrippo, editors, *Proc. 7th. Feature Interactions in Telecommunications and Software Systems*, pages 59–76, Amsterdam, Netherlands, June 2003. IOS Press.
5. Ji He and K. J. Turner. Specification and verification of synchronous hardware using LOTOS. In J. Wu, S. T. Chanson, and Q. Gao, editors, *Proc. Formal Methods for Protocol Engineering and Distributed Systems (FORTE XII/PSTV XIX)*, pages 295–312, London, UK, Oct. 1999. Kluwer Academic Publishers.
6. J. Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks*, 29:25–59, 1996.
7. K. J. Turner. Validating architectural feature descriptions using LOTOS. In K. Kimbler and W. Bouma, editors, *Proc. 5th. Feature Interactions in Telecommunications and Software Systems*, pages 247–261, Amsterdam, Netherlands, Sept. 1998. IOS Press.
8. K. J. Turner. Formalising the CHISEL feature notation. In M. H. Calder and E. H. Magill, editors, *Proc. 6th. Feature Interactions in Telecommunications and Software Systems*, pages 241–256, Amsterdam, Netherlands, May 2000. IOS Press.
9. K. J. Turner. Modelling SIP services using CRESS. In D. A. Peled and M. Y. Vardi, editors, *Proc. Formal Techniques for Networked and Distributed Systems (FORTE XV)*, number 2529 in Lecture Notes in Computer Science, pages 162–177. Springer-Verlag, Berlin, Germany, Nov. 2002.
10. K. J. Turner. Formalising graphical service descriptions using SDL. In R. Reed, editor, *SDL User Forum 03*, Lecture Notes in Computer Science, Berlin, Germany, July 2003. Springer-Verlag.
11. K. J. Turner. Representing new voice services and their features. In D. Amyot and L. Logrippo, editors, *Proc. 7th. Feature Interactions in Telecommunications and Software Systems*, pages 123–140, Amsterdam, Netherlands, June 2003. IOS Press.
12. K. J. Turner and Qian Bing. Protocol techniques for testing radiotherapy accelerators. In D. A. Peled and M. Y. Vardi, editors, *Proc. Formal Techniques for Networked and Distributed Systems (FORTE XV)*, number 2529 in Lecture Notes in Computer Science, pages 81–96. Springer-Verlag, Berlin, Germany, Nov. 2002.
13. VoiceXML Forum. *Voice eXtensible Markup Language*. VoiceXML Version 1.0. VoiceXML Forum, Mar. 2000.