Kenneth J. Turner. Analysing Interactive Voice Services
(pre-publication version), Computer Networks, 45:665-685,

Analysing Interactive Voice Services

Kenneth J. Turner

Computing Science and Mathematics, University of Stirling, Stirling FK9 4LA, UK

Abstract

IVR (Interactive Voice Response) services are increasingly prevalent in automated telephone enquiry systems. VoiceXML (Voice eXtensible Markup Language) has become one of the leading languages for IVR. The nature of IVR services is introduced, along with an explanation of how they are represented in VoiceXML. However a VoiceXML description is at a low level, so it is difficult to gain an overview of the service that is offered. There is also no rigorous way to check the integrity of an IVR application.

CRESS (Chisel Representation Employing Systematic Specification) is a graphical notation for describing services in an abstract, language-independent manner. For this paper, IVR services are described with CRESS and translated into LOTOS (Language Of Temporal Ordering Specification) for automated analysis. Because of the infinite state space, it is not practicable to formally verify the generated specifications. Instead, the focus is on more practical solutions. The properties of a specification are checked by including observer processes to monitor undesirable situations like repeatedly prompting the user for input. MUSTARD (Multiple-Use Scenario Test And Refusal Description) is introduced as a language for defining scenario-based tests of services. The approach is illustrated with sample tests of IVR services. It is seen how MUSTARD helps to build confidence in an IVR application.

The paper also introduces a feature concept for IVR, and discusses feature interaction in this context. General categories of IVR feature interaction are presented. It is shown how CRESS and MUSTARD combine to help discover interactions among IVR features.

Key words: Feature, IVR (Interactive Voice Response), LOTOS (Language Of Temporal Ordering Specification), Service, VoiceXML (Voice eXtensible Markup Language)

Email address: kjt@cs.stir.ac.uk (Kenneth J. Turner).

1 Introduction

1.1 Interactive Voice Response

IVR (Interactive Voice Response) services have been developed during the past decade to provide a more satisfactory alternative to touch-tone systems. Touch-tone enquiry systems ('press 2 for sales') are often disliked by users due to their inflexible and crude interfaces. IVR allows users to do what they expect in a telephone call, namely to speak and to listen. IVR is convenient for users on the move, who may have little more than a mobile telephone. Although WAP (Wireless Access Protocol) is intended to provide web browsing for mobile users, it has seen only limited use. Some categories of users (e.g. the partially sighted or those without Internet access) are also disadvantaged if information is provided only via the web.

Although IVR is not new, it was initially supported by a variety of proprietary solutions. VoiceXML (Voice eXtensible Markup Language [30]) has been an important development in the standardisation of IVR. There are competing standards for IVR, but VoiceXML seems to have attracted the most support. The basic idea of VoiceXML is that users 'fill in' fields of forms by speaking in response to prompts. VoiceXML platforms usually include sophisticated support for TTS (Text To Speech, i.e. synthesised speech output) and STT (Speech To Text, i.e. speech recognition). The completed information is then typically submitted to a program or database for further processing. VoiceXML lends itself to a wide variety of applications such as news and sports information, telephone banking, sales enquiries and orders, and travel bookings. For an application such as banking, VoiceXML could provide a voice-based front-end to an existing bank system. There could also be other front-ends to the same system, e.g. for web browsing or WAP access.

1.2 Developing Interactive Voice Services

As an application of XML, VoiceXML is textual in form. However most commercial packages (e.g. Covigo Studio, Nuance V-Builder, Voxeo Designer) provide a graphical representation. VoiceXML has a nested, hierarchical structure that most packages reflect in graphical form. Some representations emphasise the relationship among VoiceXML elements, e.g. the flow of control among the fields of a form. Commercial packages are (not surprisingly) very close to VoiceXML since their aim is direct support of scripting with VoiceXML. As a programming language, VoiceXML focuses on how an IVR service is realised and not what it does. It can therefore be difficult to get a clear overview from VoiceXML of an IVR service.

It is easy, and even common, to write VoiceXML scripts that have implicit loops

and complicated logic. To some extent, VoiceXML encourages this because its form interpretation algorithm requires multiple passes through a form. The consequences of certain VoiceXML constructs may not be immediately obvious, e.g. they may cause an indefinite loop.

VoiceXML adopts a pragmatic and programmatic approach to development. There is no way to formally check or analyse a VoiceXML script. Instead, VoiceXML must be debugged using traditional software engineering methods.

VoiceXML applications are essentially single scripts, though these can be made up from a number of individual documents (i.e. files). VoiceXML supports unconditional transfers (**goto**) and subroutine-like calls (**subdialog**) to other documents. However there is no equivalent of a feature. In fact, VoiceXML does not even use the term service.

In telephony, services are often composed from self-contained features. A feature is an additional function that is triggered automatically (e.g. call diversion or call blocking). From the developer's point of view, a feature is triggered by certain conditions and is not explicitly called at some point in the call processing code. Features can therefore easily add supplementary capabilities to basic call processing. The value of features has been amply demonstrated in the IN (Intelligent Network).

CRESS (Chisel Representation Employing Systematic Specification) is a front-end for defining and formalising services. CRESS was initially based on the industrial notation Chisel developed by BellCore [1]. However, CRESS has been considerably extended since its beginnings. In particular, it supports the notion of plug-in domains: the vocabulary and concepts required for each application area are defined separately. CRESS has been demonstrated on services from the IN (Intelligent Network [24]), Internet telephony [25,27], and IVR (Interactive Voice Response [27,28]).

CRESS aims to combine the advantages of an accessible graphical notation, analysis via translation to formal languages, and realisation via translation to implementation languages. That is, the same service diagrams can be used for multiple purposes. CRESS is neutral with respect to the target language. For formal analysis, CRESS diagrams are automatically translated to LOTOS (Language Of Temporal Ordering Specification [11]) or to SDL (Specification and Description Language [13]); see [28] and [26] respectively. For implementation, CRESS diagrams are automatically translated to Perl (for SIP services) or to VoiceXML (for IVR services); see [25] and [27,28] respectively.

For IVR services, CRESS is intended to complement existing VoiceXML platforms. In particular, CRESS offers the following:

- CRESS is a platform-independent graphical notation for a substantial (but not complete) proportion of IVR applications. A CRESS service is represented at

a more abstract level than VoiceXML, making it easier to gain an overview of the service. VoiceXML is merely a target language for CRESS, so it should be possible to translate CRESS diagrams into other IVR languages.
- CRESS supports features and services. These are not directly recognised in IVR, so their addition provides useful extra capabilities. Without features, IVR applications have to explicitly call supplementary capabilities.
- It can be difficult to check whether a realistic IVR application will behave correctly in all circumstances (e.g. will not stop prematurely or loop indefinitely). Through translation to a formal language, CRESS supports rigorous analysis of IVR services. CRESS is also accompanied by a scenario-based testing language that is used to validate IVR applications. The same approach also contributes to detecting feature interactions.
- VoiceXML is not formally defined. Some concepts are only vaguely described (e.g. event handling) and some are loosely defined (e.g. the semantics of expressions and variables). Through translation to a formal language, CRESS contributes to a more precise understanding of VoiceXML.

1.3 Relationship to Other Work

Graphical notations for services are, of course, fairly common. Although it has a graphical form, SDL (Specification and Description Language [13]) is a general-purpose language that was not designed particularly to represent communications services. MSCs (Message Sequence Charts [12]) 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 has support for specific domains, and they cannot be translated into a range of languages. Perhaps surprisingly, there does not appear to have been other work on graphical or formal specification of IVR services.

As noted earlier, there are a number of commercial tools for VoiceXML. These offer rather more complete support for IVR than CRESS. However they are focused on VoiceXML only, and do not offer any kind of formal analysis. Their (graphical) representations of services are very close to VoiceXML, so they are useful only to specialists. Figure 1 is an example of what VoiceXML looks like in a commercial tool; this corresponds to the *Donation* service described by CRESS in figure 2.

Commercial VoiceXML tools do not support rigorous analysis of IVR services. The translation of CRESS into LOTOS or SDL gives formal meaning to IVR service descriptions. The translation provides access to any analytic technique based on these languages. Among these, the author's own approach [23] is one of several that might be used.

Feature interaction in telephony is a much studied issue (e.g. [7]). The basic prob-

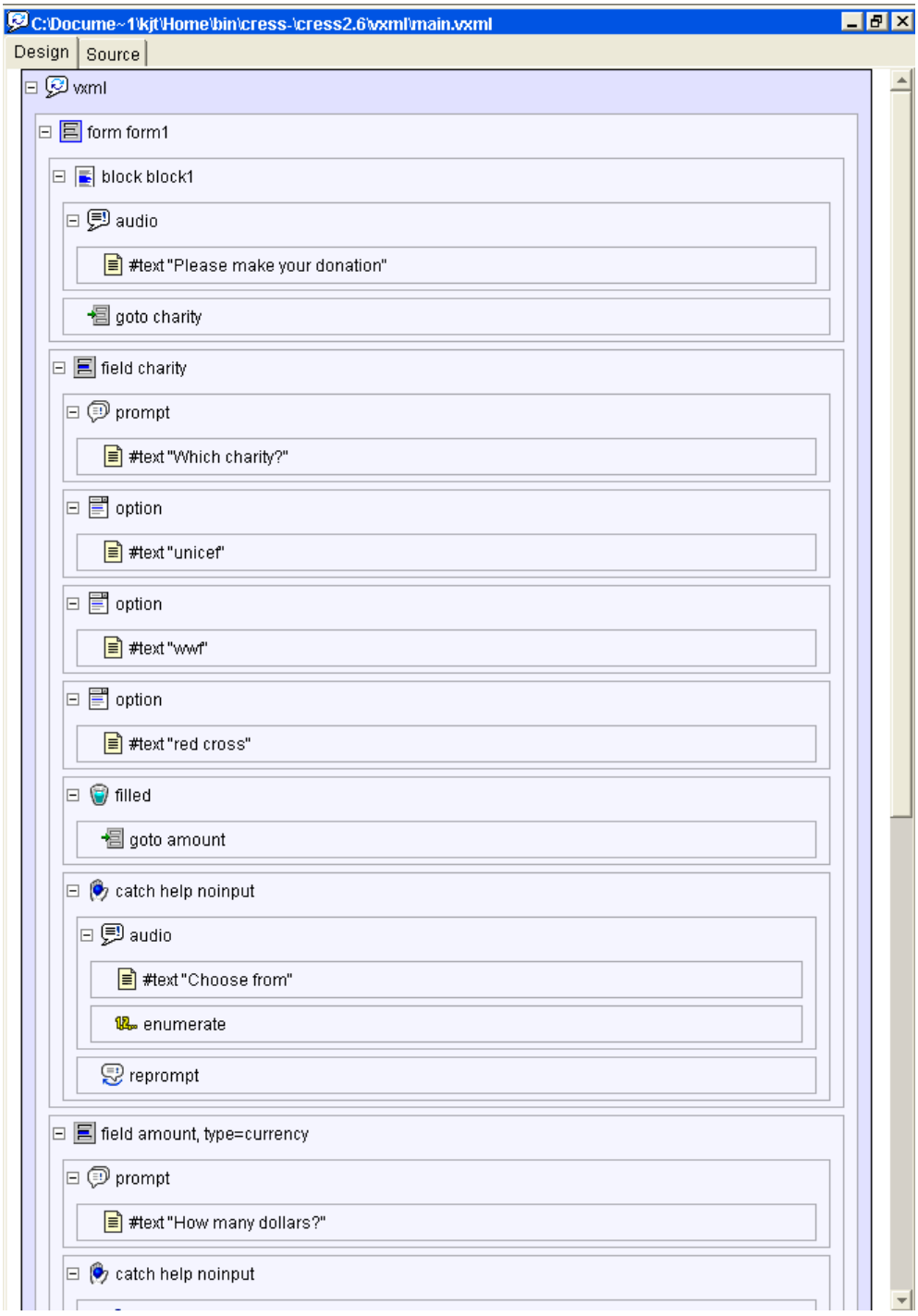

Fig. 1. Partial Screenshot of Nuance V-Builder (Version 1.2)

lem is that independently designed features can interfere with each other. It has been shown that feature interactions occur in a variety of other domains such as building control [15], email [5,9], Internet telephony [14,25], lift control [17], mobile communication [32], multimedia [4,22], policies [19], and the web [31]. The work reported here shows how feature interaction can arise with IVR.

1.4 Overview of The Paper

The new contributions made by this paper are the application of CRESS to IVR services and features, the rigorous analysis of IVR applications, and the analysis of feature interactions in IVR. Section 2 introduces IVR and its realisation using VoiceXML. Section 3 gives an overview of the CRESS notation as used to describe IVR services. Section 4 describes how IVR services are analysed, including the use of observer processes and a specialised test notation. Section 5 discusses the nature of feature interaction in IVR, and shows how CRESS can be used to discover feature interactions.

2 Interactive Voice Response using VoiceXML

2.1 Interactive Voice Response Systems

As an example of IVR, the following hypothetical dialogue might occur with a telephone banking system:

System: You have called the Automated Phone Bank.
What would you like to do?
User: *Silence*
System: You can ask for your balance, request a statement, or close your account
User: My balance please
System: What is your account number?
User: Four eight five six seven one
System: There is no account with this number, please try again
User: Four three five six seven one
System: What is the PIN for this account?
User: Five three eight one
System: Your balance is seven hundred and fifty one dollars.
Do you wish another service?
User: No thanks
System: Thank you for calling, goodbye

Commercial packages allow for a variety of natural languages to be used in both speech synthesis and speech recognition. The core logic of an IVR script can be the same, independent of the user's language. Speech synthesis is not too difficult, though it is harder to achieve acceptable emphasis and intonation. Text to be spoken by an IVR system is usually annotated to indicate these aspects. The pronunciation of unusual words and phrases can be also given using the markup grammar. Although synthesis can produce natural-sounding speech, serious applications usually make use of pre-recorded human speech.

Speaker-independent speech recognition is very difficult except for limited vocabularies. (An IVR system must be able to deal with any caller.) Where IVR gains is that the system does not need to recognise unpredictable speech. In answer to the first banking question above, for example, the system needs to deal with only a limited variety of replies. The grammar used to define these can allow for reasonable variations such as 'current balance' or 'check my balance'. The speech recogniser will choose the best match between the user's input and the grammar. Pre-defined grammars deal with common input formats such as currency amounts, dates, pure numbers, telephone numbers, times and yes/no answers. Script-defined grammars allow arbitrary inputs using BNF-like definitions.

IVR applications can usually invoke external programs, such as scripts in other languages, calls to web server pages, or code to interrogate a database.

2.2 VoiceXML Scripts

VoiceXML (Voice eXtensible Markup Language [30]) is the major language used to write IVR applications. VoiceXML has built on earlier languages for IVR. To give an idea of VoiceXML, the following is a simplified extract of what the telephone banking application might look like.

```
<?xml version='1.0'?>                                  <!-- XML -->
<!DOCTYPE vxml PUBLIC>                          <!-- VoiceXML definition -->
<vxml version='1.0'>                                   <!-- VoiceXML -->
<form>                                                      <!-- form -->
  <property name='timeout' value='3'/>          <!-- input timeout 3 secs -->
  <block>                                        <!-- start-of-form code -->
    <audio>                                              <!-- audio output -->
      You have called the Automated Phone Bank.
    </audio>
  </block>
  <catch event='nomatch' count='3'>                 <!-- third no match? -->
    <audio>                                              <!-- audio output -->
      Sorry - too many attempts, goodbye
    </audio>
```

```
  </catch>
  <field name='action'>                          <!-- field for action -->
    <prompt>                                     <!-- audio prompt -->
      What would you like to do?
    </prompt>
    <option>balance</option>                     <!-- balance option -->
    <option>statement</option>                   <!-- statement option -->
    <option>close</option>                       <!-- close option -->
    <filled>                                     <!-- field completed -->
      <if cond='action == "balance"'/>           <!-- balance action? -->
        <goto next='#balance'/>                  <!-- to balance form -->
      <elseif cond='action == "statement"'/>     <!-- statement action? -->
        <goto next='#statement'/>                <!-- to statement form -->
      <else/>                                    <!-- close action -->
        <goto next='#close'/>                    <!-- to close form -->
      </if>
    </filled>
    <catch event='help noinput'>                 <!-- help or no input? -->
      <audio>                                    <!-- audio output -->
        You can ask for your balance, request a statement, or close your account
      </audio>
      <reprompt/>                                <!-- repeat form input -->
    </catch>
  </field>
  ...
</form>
</vxml>
```

A VoiceXML document can contain one or more forms (or menus). At various points, including the start of a **form**, a **block** can be used for executable code such as **audio** output and assignments. A **property** is a platform-defined variable such as the *timeout* for user input. A VoiceXML script can **catch** events such as **nomatch** (unrecognised user input) subject to an optional condition or retry *count*. A **field** with simple alternatives can list its **option** values. A **prompt** requests user input. Once a field has been **filled** (the user has provided matching input), the value assigned to its field variable can be used for further processing. In the above, the value of *action* is used to determine the URL to **goto**; the '#' notation is borrowed from HTML to reference a label. If the field is not filled in as expected, a **help** event (the user asked for assistance) or a **noinput** event (the user did not speak) is caught and used to trigger a **reprompt**. This has the effect of scanning the form from the top, usually causing the most recent field to be prompted for again.

VoiceXML supports a hierarchical event model. Event handlers may be defined at four levels: platform, application, form, field. Platform handlers provide fall-back support, though they are usually too general to be useful. Application handlers gov-

ern 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 to a matching handler. As well as those mentioned above, standard events include **cancel** (the user cancelled processing), **error** (a run-time error occurred), and **exit** (the user asked to exit). 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. *balance.failure.PIN*). Normally this would be caught by a handler for this exact event. But if there is nothing to match, a handler for *balance.failure* (or failing that *balance*) may deal with the event. If no handler matches, the application terminates.

Events are also implicitly associated with a prompt count that is incremented each time a field is entered. This may be used to 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, e.g. because several handlers could apply. VoiceXML does not define what happens if conditions overlap – in fact the behaviour is nondeterministic.

In order to interwork with a web server, a VoiceXML script can **submit** values to a URL. This may return dynamically-created VoiceXML (e.g. to announce the result) that allows execution to continue. VoiceXML can also enter an embedded **script** that uses ECMASCRIPT (JAVASCRIPT) to perform arbitrary computations. In fact, VoiceXML shares variables and expressions with ECMASCRIPT. A VoiceXML document can **goto** another one, or can call it as a **subdialog** (like a subroutine).

3 Formalising Interactive Voice Response Services

3.1 The CRESS Notation

CRESS is a graphical notation for describing the possible behaviour of a service. State is intentionally implicit in CRESS because this allows more abstract descriptions to be given. Arcs between states may be guarded by event conditions or by value conditions. 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.

Ultimately, CRESS deals with a single diagram. However it is convenient to con-

struct 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** *donate.jsp ″amount charity″*). 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. A node number is followed by '!' to prevent feature templates from matching the node. 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* $\leftarrow$ *3*).

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 conditions 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** *charity, amount*). A rule box may define configuration information like parent diagrams, chosen features and translator options. Definitions can be given of macros with optional parameters. Rule boxes have yet other uses [24,25,27] that are not so applicable to IVR.

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 usually 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.

As an example of CRESS for IVR, suppose the imaginary Charities Bank requires a service for telephone donations to charity. Figure 2 shows the CRESS root diagram for this sample application. This defines the application variables *charity* (UNICEF, WWF, Red Cross) and *amount* (the donation in US dollars). A *Welcome* message is defined as a macro for general use.

The root application asks the caller to state the charity and the amount, for submission to the *donate.jsp* web page. If the user asks for help or says nothing following a prompt, an explanation is given and the user is reprompted. A currency amount is read as a string whose first three characters give the currency code (e.g. ″USD″). If the user says another currency (e.g. ″UKL″ means pounds sterling), the user is reprompted for the amount. **Retry** in node 7 first clears the value entered for *amount*, otherwise the field would be ignored on the reprompt because it has already been filled.

Speech output may contain markup. Variable values are interpolated as, for example, *$charity*. The current options list is interpolated with *$enumerate*. Speech is emphasised with *$emph(text)*. The pronunciation of a word may be given with *$sub(loch,lough)* that substitutes the first word for the second. Text may be spoken according to a particular class of expression, e.g. *$class(phone,467423)* or *$class(number,467423)*.

Suppose that Charities Bank has a range of applications besides the donation application in figure 2. There might, for example, be separate applications to enquire what charities are supported, to ask for a statement of the donations made to date, or to request a tax relief statement. It would be desirable to ensure a consistent treatment of all these applications. For example, 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.

Figure 3 is the *Introduction* feature that defines an introductory environment for all Charities Bank applications. The feature is placed just after the **Start** node in the root diagram (as indicated by the '+' after the triggering node number); a **Start** node is implicit prior to figure 2 node 1. Introductory messages (including that of the *Welcome* macro) 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 3 ensures that after three such failures the user is disconnected. Figure 3 also defines a platform property: here the timeout for no input is set to three seconds (*timeout* $<-$ *3*).

Although the *Introduction* feature defines a specific input timeout, it could be useful to have a feature that disables timeouts for any application. Figure 4 defines the *Wait*

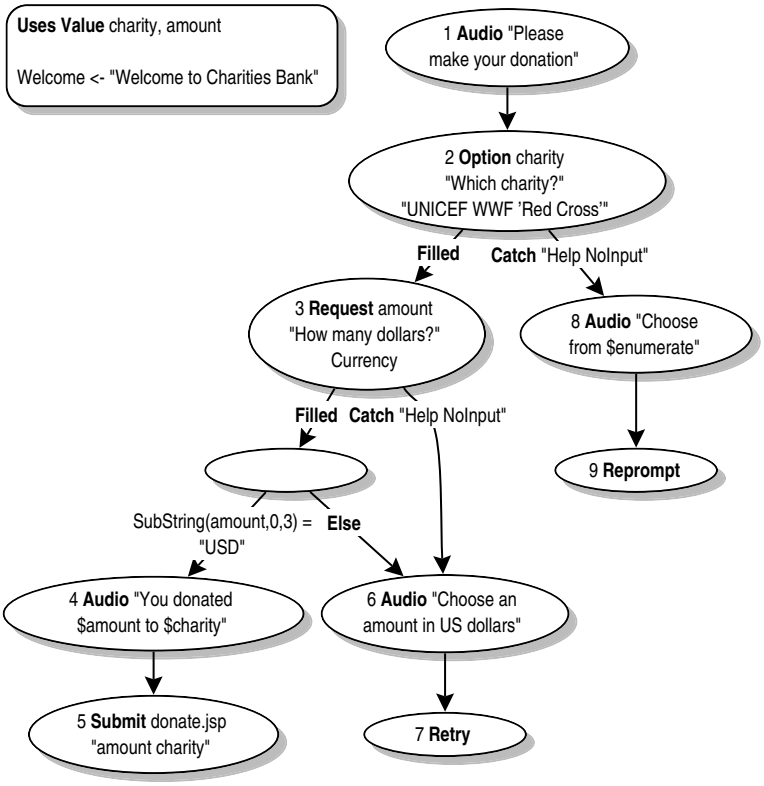

Fig. 2. CRESS Root Diagram for Charity Donation Application

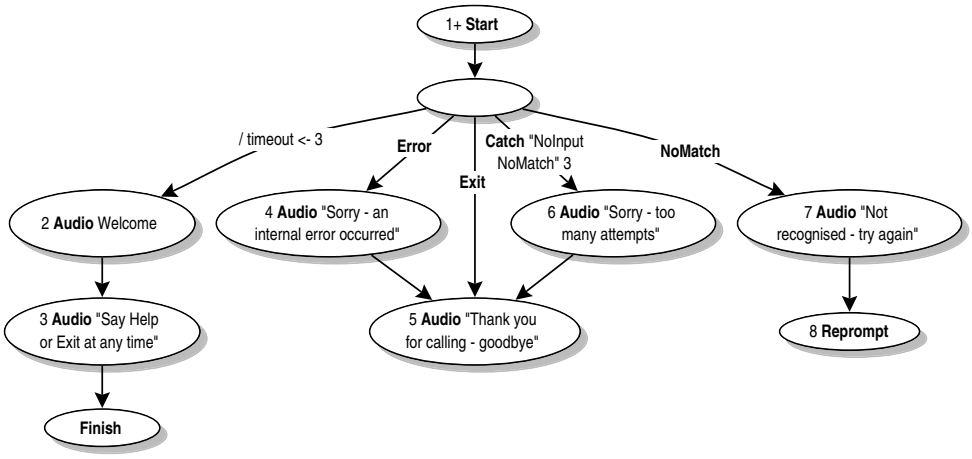

Fig. 3. CRESS Feature Diagram to introduce Charities Bank Applications

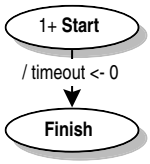

Fig. 4. CRESS Feature Diagram to disable Input Timeout

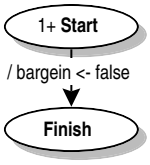

Fig. 5. CRESS Feature Diagram to disable Prompt Barge-In

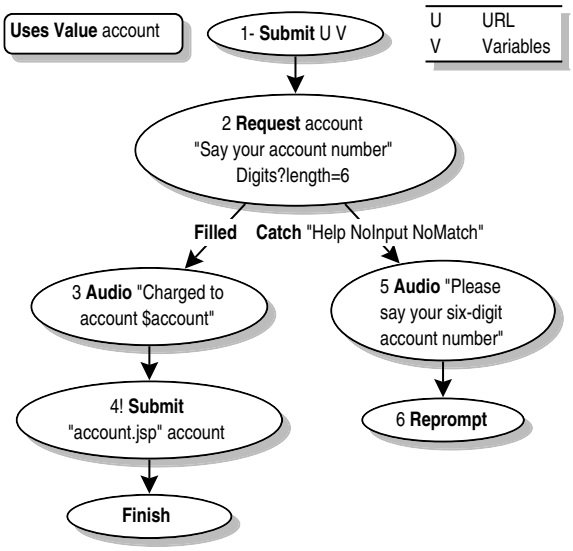

Fig. 6. CRESS Feature Diagram for Account

feature that sets *timeout* to zero. Figure 5 is also a generic feature that disables user barge-in, i.e. interruption of a prompt.

Figure 6 defines the *Account* feature that asks the user to supply an account number. This feature could be generic, and not just for the Charities Bank. The feature is triggered before other information is submitted to a web server, i.e. the charity and amount in the case of a donation. The trigger is a **Submit** action (node 1), being executed just before it (as indicated by the '–' after the triggering node number). The account number is submitted to the *acount.jsp* web page. Help is provided as required for the *account* field. The digit string grammar may define a specific length (6 for *account*), or may define a minimum and maximum number of digits.

Figure 7 is a similar feature *PIN* that asks the user for the Personal Identification Number to access an account. As an example of speech markup, *$sub* is used to say PIN as a word rather than as P-I-N.

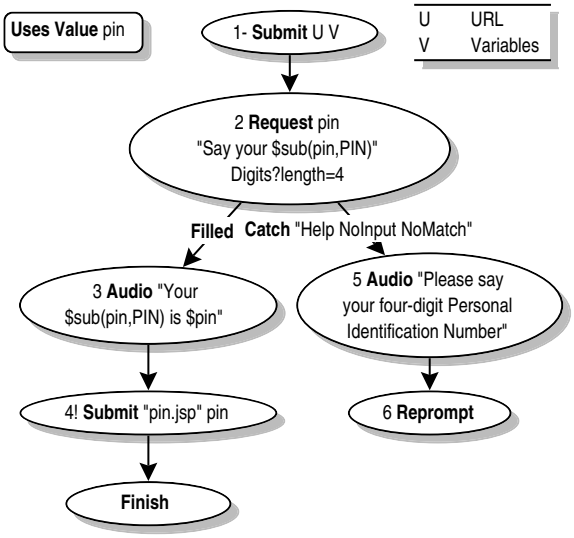

Fig. 7. CRESS Feature Diagram for PIN

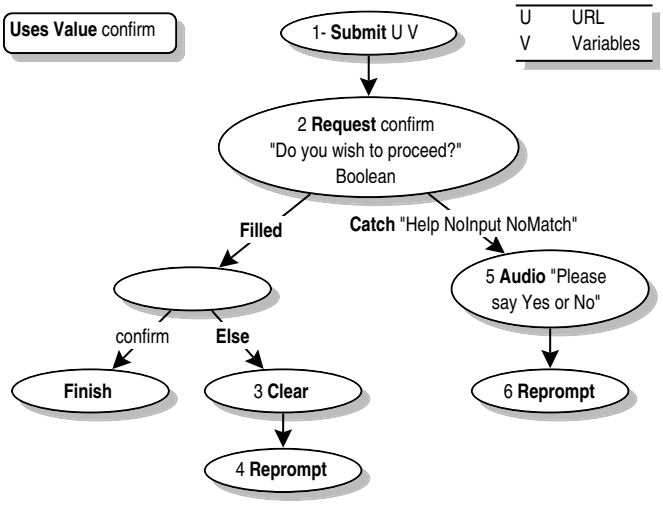

Fig. 8. CRESS Feature Diagram for Confirmation

Figure 8 is another generic feature. *Confirm* asks for confirmation before final submission of information. Since the *Account*, *PIN* and *Confirm* features are triggered by the same **Submit** action, feature priorities ensure that they are applied in this order.

3.3 Translating CRESS

The CRESS toolset is written largely in Perl for portability, comprising about 14,000 lines of code and six main tools. Including test scenarios, there are about 600 supporting files for all domains and target languages.

For IVR, CRESS diagrams are automatically translated into VoiceXML. On the

CRESS	VoiceXML
Audio *message*	**audio** *message*
Clear *variables*	**clear** with namelist *variables*
Menu *variable prompt choices*	**menu** name *variable*, **prompt**, **choice** values
Option *variable prompt options*	**field** name *variable*, **prompt**, **option** values
Prompt *message*	**prompt** *message*
Reprompt	**reprompt**
Request *variable prompt grammar*	**field** name *variable*, type *grammar*, **prompt**
Retry	Undefine current **field** variable, **reprompt**
Submit *URL variables*	**submit** to *URL* the namelist *variables*

Fig. 9. CRESS-VoiceXML Correspondence

whole, the translation is straightforward. Figure 9 shows the main correspondence between CRESS and VoiceXML. Forms (and menus) are prominent in VoiceXML since this focuses on structural issues. However CRESS emphasises the flow of control, and so does not give the same prominence to them. Instead, fields are introduced implicitly (with **Menu**, **Option** and **Request**). The flow of control in VoiceXML can be complex and implicit, e.g. a reprompt goes back to the start of a form and then selects the first available unfilled field. A CRESS diagram can reflect the same implicit flow, or can indicate this explicitly. CRESS includes the **Retry** action for re-inputting the current field – something that VoiceXML does not directly support.

The general principles for formalising CRESS appear in [24]. Broadly speaking, inputs and outputs are treated the same in LOTOS: they are translated as events (usually) or as process calls (where paths converge on a node). Inputs require special treatment in SDL as they are permitted only at the beginning of a state transition. Alternative inputs of the same signal or variable need a complex translation. Outputs are converted straightforwardly into SDL.

For IVR services specifically, translation into LOTOS and SDL is described in [28] and [26] respectively. The major aspect of IVR that needs a specialised translation is event handling. Although this occurs dynamically in IVR since event names can be constructed during execution, event dispatching needs to be defined statically in LOTOS or SDL. Fortunately, it is possible to determine the hierarchy of event handlers at translation time. The event dispatcher reflects this hierarchy, passing

an event to the relevant process (LOTOS) or label (SDL). Actions such as those in figure 9 are domain-dependent, so their translation into LOTOS or SDL is specific to IVR. Speech is not, of course, rendered directly in LOTOS or SDL but as strings carried in event parameters.

4 Analysing Interactive Voice Response Services

4.1 Analysis in General

An IVR application can be executed like any script. Some commercial packages allow VoiceXML to be run in an offline IDE, while others require the script to be run by an online environment. In either case, debugging follows typical programming practice. This is, of course, time-consuming and risks undetected errors. Since CRESS diagrams can be translated into LOTOS and SDL, this offers new possibilities for automated analysis. For illustration, this paper concentrates on what can be done with LOTOS.

There is, of course, nothing special about the LOTOS generated by CRESS. Any standard LOTOS technique can be used for validation such as step-by-step simulation, symbolic execution, rapid prototyping and testing. Any standard LOTOS technique can be used for verification such as equivalence checking, model checking and theorem proving.

The main problem in formal analysis of the generated LOTOS is that the state space is usually infinite in two ways: events carry infinite sorts (such as speech strings), and behaviour may recurse indefinitely. As a result, verification is impracticable using standard temporal logics like CTL (Computation Tree Logic [3]) or LTL (Linear Temporal Logic [18]), and standard model-checkers like SPIN [10].

A number of solutions might be adopted:

- A symbolic transition system could be generated, allowing analysis of a finite transition system using symbolic model-checking. Although [6] is a promising basis for this, the work has not yet progressed to the point where automated analysis is feasible.
- Symbolic on-the-fly test generation could be used [20], though again this technique is not yet usable for IVR services.
- Event values could be restricted when the state space is generated. The Parameter Constraint Language of [29] allows the specifier to define interesting parameter values as annotations on events. These are translated into parallel constraint processes that limit the state space. The programming interface for CADP (Cæsar Aldébaran Development Package [8]) also allows the specifier to selectively enu-

merate sorts during state space generation. Unfortunately neither approach is particularly suitable for IVR services since the restriction on values is context-dependent.

- Observer processes can be placed in parallel with the main behaviour to check for undesirable conditions. The state space can then be generated, stopping if an observer process forces deadlock due to violation of a required property. This is a practical solution that is explored in section 4.2.
- Validation (i.e. testing) can be used to check correct behaviour. Since the tests are concrete and finite, validation is a practical solution that is explored in sections 4.3 and 4.4.

4.2 Verification using Observer Processes

The idea of observer processes is well established. In fact, the approach resembles that of model-checking except that properties are formulated as observer processes.

The LOTOS specification generated by CRESS for VoiceXML has the user and a web server as its environment. The top-level structure of the behaviour is as follows:

```
Hide Reco In                                    (* hide recogniser signals *)
  (
    Application [Reco,Serv,User]                (* VoiceXML application *)
  |[Reco]|                                      (* synchronised on recogniser messages *)
    Recogniser [Reco,User]                      (* VoiceXML recogniser *)
  )
||                                              (* synchronised on all messages *)
  Observer [Reco,Serv,User]                     (* VoiceXML observer *)
```

The *Observer* process synchronises on all events at gates *Reco* (messages to/from the speech recogniser), *Serv* (messages to the web server) and *User* (messages to/from the user). By default, the *Observer* process permits any events at these gates. This requires care to make sure that accidental deadlocks are avoided. Fortunately the variety of event structures is small, so it is practicable to ensure that all possible events are handled.

Specific conditions can be checked with observer processes. For example, an easy mistake with an IVR application is to loop indefinitely instead of giving up after a certain number of user attempts. The following observer process counts how many times the same **Request** for input is repeated. If it reaches a certain limit, the behaviour deadlocks. Since all observer processes are synchronised with the main behaviour, the whole specification deadlocks at this point.

```
Process RecogniserPrompt [Reco]                 (* repeated prompt up to limit *)
 (prompt:Text, count,limit:Nat) : Exit :=
   Reco !Request ?promptNew:Text ?grammar:Grammar   (* allow request ... *)
```

```
    [(promptNew Eq prompt) Implies (count Lt limit)];     (* to limit if same *)
  (
    [promptNew Eq prompt] ->                                (* same prompt? *)
      RecogniserPrompt [Reco] (prompt, count + 1, limit)      (* incr. count *)
  []
    [promptNew Ne prompt] ->                              (* different prompt? *)
      RecogniserPrompt [Reco] (promptNew, 1, limit)        (* set count to 1 *)
  )
EndProc
```

Many variations on this are used. **Menu** and **Option** inputs are checked in the same way. A limit can be placed on the total number of prompts in any one session, and not just on repetitions of the same prompt. Relationships between events can also be checked, for example to ensure that there is no **Submit** to the web server if the user invokes **Cancel** or **Exit**.

The state space is generated up to a certain depth for the specification including observer processes. If any deadlocks are found, the trace up to that point is used to identify the cause of the problem. For example, if a prompt count limit is met it can be seen which field is being repeatedly requested.

Consider the *Donation* application in figure 2. Does it contain an infinite loop? An observer process like *RecogniserPrompt* above was defined to check for repeated **Option** inputs beyond a limit of 3. The specification was then explored to a depth of 15 using an *Expand* function of LOLA (LOTOS Laboratory [16]). This discovers several deadlocking traces such as the one below. (For readability, text strings are given rather than their actual, rather ugly representation in LOTOS.)

```
User !Audio !″Please make your donation″;      (* get introductory message *)
User !Audio !″Which charity?″;                       (* get charity prompt *)
User !Tone ?input:Text;                     (* provide invalid touch tone *)
User !Audio !″Input was not recognised″;             (* get error message *)
User !Audio !″Which charity?″;                       (* get charity prompt *)
User !Event ?eventEvent;                      (* cause invalid user event *)
User !Audio !″Input was not recognised″;             (* get error message *)
User !Audio !″Which charity?″;                       (* get charity prompt *)
User !Voice ?input:Text;                          (* select invalid charity *)
User !Audio !″Input was not recognised″;             (* get error message *)
```

This sequence arises because the user gives three incorrect inputs in a row. Now that the prompt has been repeated up to the limit, it is no longer permitted and the specification deadlocks. In fact, figure 2 allows this prompt to be re-issued indefinitely. The application needs to be modified to prevent this.

Use of observer processes requires LOTOS to be written. However the goal of CRESS is to allow non-specialists to define and investigate services. It is thus necessary to hide the underlying formalism. Desirable properties of a service should be expressed using a neutral language. The idea is to characterise the expected behaviour of a service using scenarios, much as use-case scenarios are used in software engineering. Of course, a scenario-based approach can achieve only limited validation of a service. However it is practical (even for an infinite state space) and conforms to software engineering practice.

CRESS is therefore complemented by its culinary counterpart MUSTARD (Multiple-Use Scenario Test And Refusal Description). As the name suggests, MUSTARD is used to define scenario-based tests of what a service must do. However, as is common in testing it is also important to check what a service must *not* do: its refusals. Refusal-based testing is more stringent in checking that a specification is not too loose in its behaviour. MUSTARD is an elaboration of the ANTEST language developed for ANISE (Architectural Notions In Service Engineering [23]).

Similar to CRESS, MUSTARD is used to formulate tests independently of the actual language used for testing. MUSTARD must therefore be translated into a particular test realisation. In the work reported here this is LOTOS, though MUSTARD should be capable of translation into MSCs (for use with SDL), TTCN (Tree and Tabular Combined Notation), etc. Although MUSTARD could have been a graphical notation like CRESS, the requirements for expressing tests are quite different from those for expressing services. MUSTARD was therefore designed as a textual language that emphasises combinations of simpler sub-tests.

The MUSTARD translator is mainly written in the *M4* macro language [21], with a Perl wrapper that automates the validation procedure. Each service or feature is associated with a MUSTARD file that defines the tests to be performed on it. This allows all services and features to be validated with a single command. For example, the *Confirm* feature in figure 8 is validated as follows using the *TestExpand* function of LOLA:

Testing Confirm Accept ...	Pass	1 succ	0 fail	0.9 secs
Testing Confirm Incorrect ...	Pass	1 succ	0 fail	0.4 secs
Testing Confirm Retry ...	Pass	1 succ	0 fail	0.4 secs

A test ought to Pass. A scenario might legitimately be passed in multiple ways, in which case the number of succ(essful) paths is reported. If all possible paths lead to failure, this is reported as Fail. If some paths are successful and some are not (usually due to non-determinism), the test is reported as Inconclusive. In the case of a refusal test, success means that the test does not perform the undesirable behaviour.

If a test has one or more failure paths, these are reported. However, failure is discovered in LOTOS terms. The LOTOS traces are therefore be translated back into the MUSTARD notation before being reported. This preserves the language independence of the approach.

The tests formulated using MUSTARD are used to check the specification of an IVR application, determining if it has unexpected or undesirable behaviour. The same tests can also be used as scripts to evaluate the live IVR application, e.g. after compilation of the CRESS description into VoiceXML. This is potentially more systematic than the 'Wizard of Oz' procedures that are often used, wherein an expert tester exercises the behaviour of the live application.

4.4 The MUSTARD Test Notation

Figure 10 summarises the MUSTARD test notation. The simplest behaviour involves the environment (user or web server) receiving a signal (**recv**) or sending one (**send**). More complex tests combine these. At the topmost level, a successful test leads to the internal success event (*OK*). For example, a test of the *Donation* behaviour in figure 2 might be:

```
test(Donation1,                                          % donation test
  succeeds(                                              % successful sequence
    recv(Audio,Please make your donation),               % get introductory message
    recv(Audio,Which charity?),                          % get charity prompt
    send(Voice,WWF),                                     % select WWF
    recv(Audio,How many dollars?),                       % get amount prompt
    send(Voice,$50),                                     % select $50
    recv(Audio,You donated $50 to WWF),                  % get confirmatory message
    recv(Server,donate.jsp,$50,WWF)))                    % get server request
```

A complete test like this could be overly prescriptive because it insists on behaviour that is not crucial to the test. This is particularly important for a feature, where the focus should be on testing the feature and not its surrounding behaviour. For this reason, **wait** is provided as an alternative to **recv** so that other behaviour is ignored until the required one. The above test, for example, could be simplified to its basic elements:

```
test(Donation2,                                          % donation test
  succeeds(                                              % successful sequence
    wait(Audio,Which charity?),                          % get charity prompt
    send(Voice,WWF),                                     % select WWF
    wait(Audio,How many dollars?),                       % get amount prompt
    send(Voice,$50),                                     % select $50
    wait(Server,donate.jsp,$50,WWF)))                    % get server request
```

MUSTARD	Meaning
% *text*	comment
decides(*behaviour*,...)	provides behaviours as non-deterministic (system-decided) alternatives
depends(*condition*,*test*,...)	if first condition holds then do first test, else check later condition/test pairs
exits(*behaviour*,...)	executes behaviours in sequence and then exits
fails(*behaviour*,...)	executes behaviours in sequence and then stops
interleaves(*behaviour*,...)	interleaves behaviours in parallel
offers(*behaviour*,...)	provides behaviours as deterministic (user-decided) alternatives
present(*feature*)	checks if feature is present
recv(*signal*,*parameters*)	environment receives signal with given parameters
refuses(*behaviour*,...)	executes behaviours in sequence, but the last behaviour must not happen
send(*signal*,*parameters*)	environment sends signal with given parameters
sequences(*behaviour*,...)	executes behaviours in sequence
succeeds(*behaviour*,...)	executes behaviours in sequence, then causes the internal success event and stops
test(*name*,*behaviour*)	define test with given name and behaviour
wait(*signal*,*parameters*)	absorbs events, continuing on occurrence of given signal with given parameters

Fig. 10. Summary of MUSTARD Notation

In order to ground a partial test like this, it is still necessary to provide some key inputs such as the choice of charity and amount.

Suppose the *Confirm* feature in figure 8 is to be tested for the user agreeing to proceed. The following waits for the confirmation prompt, and then issues a user agreement.

```
test(Confirm1,                                        % confirmation test
 succeeds(                                            % successful sequence
     wait(Audio,Do you wish to proceed?),             % get confirm prompt
     send(Voice,Yes)))                                % agree
```

Rather than use **succeeds**, it is possible to construct a successful test from more basic behaviours. The **sequences** operator yields a sequence of steps that can then lead to success. The elements of a sequence can be compound behaviours built with other operators. A choice can be provided to the environment with **offers**. In such a case, **exits** may be used to allow a sequence to continue.

The following test of *Confirm* checks what happens if the user does not answer Yes or No. As an (artificial) example of constructing a successful sequence, an inner **sequences** is used with an outer **succeeds**. After the confirmation prompt the user may ask for help, say nothing, or say the wrong thing. The user should then be reprompted. Agreeing terminates the test successfully.

```
test(Confirm2,                                  % confirmation test
  succeeds(                                     % successful sequence
    sequences(                                  % follow sequence
      wait(Audio,Do you wish to proceed?),      % get confirm prompt
      offers(                                   % offer alternatives
        exits(send(Event,Help)),                % ask for help
        exits(send(Event,NoInput)),             % say nothing
        exits(send(Voice,Eh?))),                % say wrong thing
      recv(Audio,Please say Yes or No),         % advise user
      recv(Audio,Do you wish to proceed?),      % get confirm prompt
      send(Voice,Yes))))                        % agree
```

Other operators can be used for more complex tests. For generality concurrent tests may be formulated using **interleaves**, though this finds little use in IVR testing. A system choice can be made using **decides**, unlike **offers** which leaves the choice up to the user. Conditional tests make use of **depends**, with conditions being arbitrary boolean expressions. A common form of conditional test uses **present** to check if a given feature has been deployed. The following offers alternative tests depending on whether the *Donation* or *Order* root diagram is present.

```
test(Multiple,                                  % multiple tests
  depends(                                      % conditional dependency
    present(Donation),                          % donation root diagram?
      ...,                                      % donation test
    present(Order),                             % order root diagram?
      ...))                                     % order test
```

Refusal tests are the most stringent, as they say what must not happen. The overall behaviour is defined by **refuses**. This contains initial steps that must happen. The final (possibly composite) behaviour that must not happen is introduced by **fails**. The following allows a donation up to the point at which the user does not agree to proceed. Following this, the donation request must not be sent to the server. That is, the complete test must not allow a sequence leading to this server request.

```
test(Confirm3,                                  % confirmation test
  refuses(                                      % refusal sequence
```

```
    wait(Audio,Which charity?),                    % get charity prompt
    send(Voice,Red Cross),                         % select Red Cross
    wait(Audio,How many dollars?),                 % get amount prompt
    send(Voice,$30),                               % select $30
    wait(Audio,Do you wish to proceed?),           % get confirm prompt
    send(Voice,No),                                % disagree
    fails(                                         % failure sequence
      wait(Server,donate.jsp,$30,Red Cross)))      % get server request
```

With these building blocks, complex tests can be formulated in MUSTARD. Each root diagram or feature diagram is associated with a set of scenarios and refusals that characterise its behaviour. The tests are automatically applied when a feature is deployed, whether in isolation or in combination with other features.

As an indication of how MUSTARD is translated into LOTOS, the following is the automatic translation of test *Confirm2* above:

```
Process Confirm2 [Serv,User,OK] : NoExit :=          (* confirm test *)
  Wait1 [Serv,User]                                  (* wait for confirm prompt *)
>>                                                   (* followed by *)
  (
    User !Event !Help;                               (* user asks for help *)
    Exit                                             (* continue *)
  []                                                 (* or *)
    User !Event !NoInput;                            (* user says nothing *)
    Exit                                             (* continue *)
  []                                                 (* or *)
    User !Voice !"Eh?";                              (* user says wrong thing *)
    Exit                                             (* continue *)
  )
>>                                                   (* followed by *)
  User !Audio !"Please say Yes or No";               (* advise user *)
  User !Audio !"Do you wish to proceed?";            (* get confirm prompt *)
  User !Voice !"Yes";                                (* agree *)
  OK;                                                (* success event *)
  Stop                                               (* stop behaviour *)
Where                                                (* local definition *)
  Process Wait1 [Serv,User] : Exit :=                (* wait for confirm prompt *)
    Serv ?par1:Text ?par2:Values;                    (* ignore server message *)
    Wait1 [Serv,User]                                (* continue waiting *)
  []                                                 (* or *)
    User !Audio ?par1:Text;                          (* get audio output *)
    (
      [par1 Eq "Do you wish to proceed?"] ->         (* confirm prompt? *)
        Exit                                         (* exit local process *)
    []                                               (* or *)
```

```
        [par1 Ne ″Do you wish to proceed?″]→      (* not confirm prompt? *)
          Wait1 [Serv,User]                        (* continue waiting *)
      )
    []                                                        (* or *)
      User !Event ?par1:Event;                    (* ignore user event *)
      Wait1 [Serv,User]                           (* continue waiting *)
    []                                                        (* or *)
      User !Tone ?par1:Text;                      (* ignore user tone *)
      Wait1 [Serv,User]                           (* continue waiting *)
    []                                                        (* or *)
      User !Voice ?par1:Text;                     (* ignore user voice *)
      Wait1 [Serv,User]                           (* continue waiting *)
  EndProc (* Wait1 *)
EndProc (* Confirm2 *)
```

5 Feature Interaction in Interactive Voice Response Services

5.1 Categories of IVR Feature Interaction

It has been seen how the integrity of an IVR application can be checked through the use of observer processes and tests. The term ‘feature’ is used loosely in the following to mean any addition to the base application, as well as to mean a CRESS feature diagram. The addition of further features to an IVR application can lead to interactions in much the same way as for telephony. However the nature of interactions is rather different for IVR. The following categories of feature interactions can be identified:

(1) IVR applications can initiate phone calls, e.g. an ordering application might set up a call to a sales assistant. Such calls may suffer from the kinds of interactions known from telephony; for example call screening might interfere with call forwarding. It is also possible for the use of a database or web server to cause interactions through conflicting demands on the underlying resources. All these interactions are strictly external to IVR and are not considered further here.

(2) Platform properties may be defined hierarchically. For example, the input *timeout* defined in figure 3 may be overridden within a field by a local feature. From the user’s point of view this would be an observable change in behaviour, e.g. the input timeout might be shortened or even disabled. Other platform properties such as *bargein*, *fetchtimeout* and *nospeechtimeout* could similarly lead to conflicts.

(3) Two features may also change an application variable inconsistently, leading to differing behaviour. The goal of the *Account* feature in figure 6 is to obtain

a literal account number. A separate feature might normalise an account number, e.g. validating a check digit or setting an account number into a standard format. The resulting account number would then depend on which features were triggered and in which order.

(4) In general the outcome of feature application could depend on the order in which features are invoked. Without a defined ordering or prioritisation, features could interfere with each other.

(5) Event handlers are defined in a hierarchy. When an event occurs, the IVR interpreter looks upwards in the hierarchy for the appropriate handler. For example, consider figures 2 and 3. If there is no input in response to the *charity* prompt (figure 2 node 2), execution follows the field handler (figure 2 node 8). However after three failures to input, the generic handler will be invoked (figure 3 node 6). A consequence of this is that a feature may unexpectedly override the usual handling of an event. It may do so at either a more local or a more global level. Non-determinism could arise if a feature added an event handler whose condition overlapped with an existing event handler.

(6) Several input grammars may be active at the same time. This is particularly true for what are called mixed-mode initiative forms that do not require the user to input in a fixed order. For example, at some point the user might be allowed to give a name or a date of birth. Provided the grammars do not overlap, this is not problematic. However grammars may overlap, e.g. 2500 might be interpreted as an amount (currency grammar) or as a PIN (digit string grammar). Such a situation could arise through design of the base application, but is unlikely as the consequences would be obvious to the designer. More realistically, overlapping grammars could arise because a feature adds a new field to an existing mixed-mode form. The effect is that feature interaction would cause non-deterministic behaviour.

(7) An indirect interaction arises with responses using DTMF (Dual-Tone Multi-Frequency, i.e. touch-tone). VoiceXML allows these in place of voice input, e.g. 1 might select the first choice from a menu. By default, DTMF digits are allocated in sequence to choices. If a feature introduces another choice earlier in the menu, the numbering of later choices will be altered.

5.2 Feature Interaction Detection in CRESS

CRESS takes a conventional view of feature interaction. If a feature behaves differently in the presence of another feature, then the two are considered to interact. From a theoretical point of view, it should be checked whether the specification of *Root*$\oplus$*Feature1* agrees (in the sense of some formal equivalence) with the specification of *Root*$\oplus$*Feature1*$\oplus$*Feature2* with respect to the behaviour of *Feature1*.

As discussed in section 4.1, complete verification of IVR specifications is impracticable. CRESS must therefore rely on more pragmatic means. The relationship

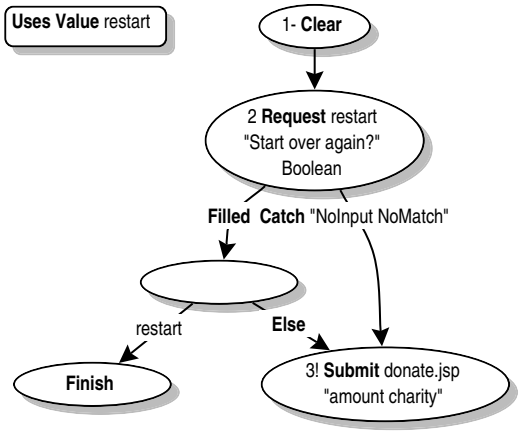

Fig. 11. CRESS Feature Diagram for Restart

checked by CRESS is that tests of *Feature1* still pass in the presence of *Feature2*. Of course this is a weak form of consistency checking, but if the scenarios reflect the key behaviour of a feature then it is reasonably thorough. Although it is common in feature interaction work to consider only pairs of features, CRESS is normally used to check for consistency in the presence of *all* other features.

The tests formulated using MUSTARD therefore play a double role: to build confidence in the correctness of an IVR service or feature, and to check for interactions among IVR features. All features are deployed with a base application, then the tests of each feature are automatically run. This is able to find problems in a reasonable timescale (minutes).

Of the categories of IVR interaction identified in section 5.1, (1) has already been ruled out of scope. Category (2) is problematic (with LOTOS and SDL at least). Timing-related properties cannot be checked without choosing a target language that supports a notion of real time. Dialogue-related platform properties can also be impossible to check, e.g. determining whether barge-in can occur requires non-atomic events. However, categories (3) to (7) all cause changes in application functionality and can be detected by CRESS.

Reconsider the Charities Bank application in section 3.2. If a feature such as *Confirm* (figure 8) clears the form, the user must re-input all information again. It may be decided that this is a drastic action that should not be taken without user confirmation. A new *Restart* feature is therefore added to prompt the user before any **Clear** action is taken. Figure 11 prompts the user for confirmation of a restart. Only if the user positively agrees does clearing take place, otherwise the donation details are submitted and the whole application exits.

This seems like a useful addition. However testing of the *Confirm* feature then reports the following:

```
Testing Confirm Accept ...          Pass      1 succ    0 fail    1.3 secs
```

Testing Confirm Incorrect ...	Pass	1 succ	0 fail	0.4 secs
Testing Confirm Retry ...	Fail	0 succ	1 fail	0.5 secs

followed by a diagnosis of the failing behaviour:

recv(Audio,Please make your donation)
recv(Audio,Which charity?)
send(Voice,Red Cross)
recv(Audio,How many dollars?)
send(Voice,$15)
recv(Audio,You donated $15 to Red Cross)
recv(Audio,Do you wish to proceed?)
send(Voice,No)
recv(Server,donate.jsp,$15,Red Cross)
<failure point>

The failing test is the following one for *Confirm*:

```
test(Retry,                                          % retry confirmation
  succeeds(                                          % successful sequence
    wait(Audio,Which charity?),                      % get charity prompt
    send(Voice,Red Cross),                           % select Red Cross
    wait(Audio,How many dollars?),                   % get dollars prompt
    send(Voice,$15),                                 % select $15
    wait(Audio,Do you wish to proceed?),             % get confirm prompt
    send(Voice,No),                                  % disagree †
    wait(Audio,Which charity?),                      % get charity prompt
    send(Voice,UNICEF),                              % select UNICEF
    wait(Audio,How many dollars?),                   % get dollars prompt
    send(Voice,$70),                                 % select $70
    wait(Audio,Do you wish to proceed?),             % get confirm prompt
    send(Voice,Yes),                                 % agree
    wait(Server,donate.jsp,$70,UNICEF)))             % get server request
```

The problem is that *Confirm* expects all form fields to be input again if the user decides not to proceed. However the *Restart* feature allows the user to cancel such an action (following the point marked † above); submission to the web server can proceed after all. As a result, the original donation data may be submitted by *Restart* in contradiction to *Confirm*. Clearly these two features interact and must be redesigned.

6 Conclusion

The nature of IVR services and their representation in VoiceXML have been explained. CRESS has been introduced as a general graphical notation for services, with particular emphasis on IVR. CRESS is formalised through translation to lan-

guages like LOTOS (the focus of this paper) and SDL. However CRESS can also be translated for implementation into languages like VoiceXML (the focus of this paper) and Perl.

CRESS offers the following benefits for IVR development:

- platform and language independence
- support of features and services
- formal definition of services
- rigorous analysis of applications.

The use of observer processes has been illustrated as a means of monitoring undesirable situations in an IVR service. The MUSTARD scenario-based test language has also been introduced with reference to IVR.

CRESS adds the concepts of service and feature to IVR. The nature of feature interaction for IVR has been discussed, including a general categorisation of the kinds of feature interaction that may arise. The use of MUSTARD to detect interactions has been explained, with an example to make the ideas more concrete.

CRESS scales satisfactorily in the following senses:

- Although a single large diagram could be drawn of an entire application, features are normally defined in their own diagrams. It is therefore practicable to handle many features.
- Validation using observer processes is modular when the processes monitor independent conditions. However a composite observer process is needed when conditions depend on shared events.
- Validation using scenario-based tests is modularised through scenarios for each individual feature. Scenarios have to be linked only when they depend on the presence of other features. Validating scenarios for one feature does not incur much of a performance penalty in the presence of multiple other features.
- Validating an IVR application normally requires only one user. However, checking for telephony interactions (e.g. among IN features) requires multiple users. Validation time then depends on the square of the number of users (which is an acceptable degree of variation).

CRESS has now proven itself in three domains: Intelligent Networks, Internet telephony, and now IVR. It has shown itself to be flexible, expressive, and able to support feature description and interaction detection.

Acknowledgements

Nuance Corporation kindly provided an academic licence for use of Nuance V-Builder ^{TM} in this work.

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. IOS Press, Amsterdam, Netherlands, May 2000.

[3] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[4] L. Blair and J. Pang. Feature interactions – Life beyond traditional telephony. In M. H. Calder and E. H. Magill, editors, *Proc. 6th. Feature Interactions in Telecommunications and Software Systems*, pages 83–93. IOS Press, Amsterdam, Netherlands, May 2000.

[5] M. Calder and A. Miller. Generalising feature interactions in email. In D. Amyot and L. Logrippo, editors, *Proc. 7th. Feature Interactions in Telecommunications and Software Systems*, pages 187–204. IOS Press, Amsterdam, Netherlands, June 2003.

[6] M. Calder and C. E. Shankland. A symbolic semantics and bisimulation for full LOTOS. In M. Kim, B. Chin, S. Kang, and D. Lee, editors, *Proc. Formal Techniques for Networked and Distributed Systems (FORTE XIV)*, pages 184–200. Kluwer Academic Publishers, London, UK, Sept. 2001.

[7] 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.

[8] J.-C. Fernández, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP (CÆSAR ALDÉBARAN Development Package): A protocol validation and verification toolbox. In R. Alur and T. A. Henzinger, editors, *Proc. 8th. Conference on Computer-Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 437–440. Springer-Verlag, Berlin, Germany, Aug. 1996.

[9] R. J. Hall. Feature interactions in electronic mail. In M. H. Calder and E. H. Magill, editors, *Proc. 6th. Feature Interactions in Telecommunications and Software Systems*, pages 67–82. IOS Press, Amsterdam, Netherlands, May 2000.

[10] G. Holzmann and D. Peled. The state of SPIN. In *Proc. 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 385–389, Berlin, Germany, 1996. Springer-Verlag.

[11] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*. ISO/IEC 8807. International Organization for Standardization, Geneva, Switzerland, 1989.

[12] ITU. *Message Sequence Chart (MSC)*. ITU-T Z.120. International Telecommunications Union, Geneva, Switzerland, 2000.

[13] ITU. *Specification and Description Language*. ITU-T Z.100. International Telecommunications Union, Geneva, Switzerland, 2000.

[14] J. Lennox and H. Schulzrinne. Feature interaction in internet telephony. In M. H. Calder and E. H. Magill, editors, *Proc. 6th. Feature Interactions in Telecommunications and Software Systems*, pages 38–50. IOS Press, Amsterdam, Netherlands, May 2000.

[15] A. Metzger and C. Webel. Feature interaction detection in building control systems by means of A formal product model. In D. Amyot and L. Logrippo, editors, *Proc. 7th. Feature Interactions in Telecommunications and Software Systems*, pages 105–121. IOS Press, Amsterdam, Netherlands, June 2003.

[16] S. Pavón Gomez, D. Larrabeiti, and G. Rabay Filho. LOLA user manual (version 3R6). Technical report, Department of Telematic Systems Engineering, Polytechnic University of Madrid, Spain, Feb. 1995.

[17] M. C. Plath and M. D. Ryan. Plug-and-play features. In K. Kimbler and W. Bouma, editors, *Proc. 5th. Feature Interactions in Telecommunications and Software Systems*, pages 150–164. IOS Press, Amsterdam, Netherlands, Sept. 1998.

[18] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[19] S. Reiff-Marganiec and K. J. Turner. A policy architecture for enhancing and controlling features. In D. Amyot and L. Logrippo, editors, *Proc. 7th. Feature Interactions in Telecommunications and Software Systems*, pages 239–246. IOS Press, Amsterdam, Netherlands, June 2003.

[20] V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *Proc. Integrated Formal Methods 00*, number 1945 in Lecture Notes in Computer Science, pages 338–357. Springer-Verlag, Berlin, Germany, Nov. 2000.

[21] R. Seindal. GNU *m4* (version 1.4). Technical report, Free Software Foundation, 1997.

[22] S. Tsang, E. H. Magill, and B. Kelly. An investigation of the feature interaction problem in networked multimedia services. In *Proc. 3rd. IEEE Communication Networks Symposium*, pages 58–61. Institution of Electrical and Electronic Engineers Press, New York, USA, July 1996.

[23] 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. IOS Press, Amsterdam, Netherlands, Sept. 1998.

[24] 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. IOS Press, Amsterdam, Netherlands, May 2000.

[25] 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.

[26] K. J. Turner. Formalising graphical service descriptions using SDL. In R. Reed and J. Reed, editors, *SDL 2003*, number 2708 in Lecture Notes in Computer Science, pages 183–202. Springer-Verlag, Berlin, Germany, July 2003.

[27] 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. IOS Press, Amsterdam, Netherlands, June 2003.

[28] K. J. Turner. Specifying and realising interactive voice services. In H. König, M. Heiner, and A. Wolisz, editors, *Proc. Formal Techniques for Networked and Distributed Systems (FORTE XVI)*, number 2767 in Lecture Notes in Computer Science, pages 15–30. Springer-Verlag, Berlin, Germany, Sept. 2003.

[29] 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.

[30] VoiceXML Forum. *Voice eXtensible Markup Language*. VoiceXML Version 2.0. VoiceXML Forum, Jan. 2003.

[31] M. Weiss. Feature interactions in web services. In D. Amyot and L. Logrippo, editors, *Proc. 7th. Feature Interactions in Telecommunications and Software Systems*, pages 149–156. IOS Press, Amsterdam, Netherlands, June 2003.

[32] P. Zave and M. Jackson. New feature interactions in mobile and multimedia telecommunications services. In M. H. Calder and E. H. Magill, editors, *Proc. 6th. Feature Interactions in Telecommunications and Software Systems*, pages 51–66. IOS Press, Amsterdam, Netherlands, May 2000.