Kenneth J. Turner. Representing New Voice Services and Their Features. In Daniel Amyot and Luigi Logrippo, editors, Proc. Feature Interactions in Telecommunication Networks VII, 123-140, IOS Press, Amsterdam, June 2003.

Representing New Voice Services and Their Features

Kenneth J. Turner
Computing Science and Mathematics, University of Stirling, Scotland
Email kjt@cs.stir.ac.uk

Abstract.

New voice services are investigated in the fields of Internet telephony (SIP – Session Initiation Protocol) and interactive voice systems (VoiceXML – Voice Extended Markup Language). It is explained how CRESS (Chisel Representation Employing Systematic Specification) can graphically represent services and features in these domains. CRESS is a front-end for detecting feature interactions and for implementing features. The nature of service architecture and feature composition are presented. CRESS descriptions are automatically compiled into LOTOS (Language Of Temporal Ordering Specification) and SDL (Specification and Description Language), allowing automated analysis of service behaviour and feature interaction. For implementation, CRESS diagrams can be compiled into Perl (for SIP) and VoiceXML. The approach combines the benefits of an accessible graphical notation, underlying formalisms, and practical realisation.

1 Introduction

1.1 Motivation

The representation of services has been well investigated for traditional telephony and the IN (Intelligent Network). Feature interaction in these domains is also well researched. However the world of communications services has moved rapidly beyond these into new applications such as mobile communication, web services, Internet telephony and interactive voice services.

This paper concentrates on developments in new voice services. Specifically, it addresses Internet telephony with SIP (Session Initiation Protocol [15]) and interactive voice services with VoiceXML (Voice Extended Markup Language [5]). Such developments have been mainly driven by commercial and pragmatic considerations. Research and theory for their services have lagged behind practice. For these new application areas, the work reported here addresses questions like:

- What is a service, and how might it be represented?
- What service architecture is needed, and what does feature composition mean?
- How can services and features be analysed and implemented?
- What properties should services have, and how does feature interaction manifest itself?

1.2 Relationship to Other Work

The author's approach to defining and analysing services is a graphical notation called CRESS (Chisel Representation Employing Systematic Specification). CRESS was originally based on the Chisel notation developed by BellCore [1]. The author was attracted by the simplicity, graphical form, and industrial orientation of Chisel. However, CRESS has considerably advanced from its beginnings. Although it lives naturally in the communications world, CRESS is not tied to this. Indeed, CRESS supports the notion of plug-in domains. That is, the vocabulary used to talk about services is defined in a separate and modular fashion. Applying CRESS to a new application mainly needs a new vocabulary for events, types and system variables.

CRESS is also neutral with respect to the target language. It can, for example, be compiled into LOTOS (Language Of Temporal Ordering Specification [7]) and SDL. This gives formal meaning to services defined in CRESS, and allows rigorous analysis of services. For direct implementation, CRESS can also be compiled into SIP CPL (Call Processing Language), SIP CGI (Common Gateway Interface, realised in Perl) and VoiceXML.

CRESS is thus a front-end for defining, analysing and implementing services. It is not in itself an approach for detecting feature interactions. Rather it supports other detection techniques. LOTOS and SDL, for example, have been used in several approaches to detecting interactions. Among these the author's own approach is applicable [16], but so are a number of others like [4, 6]. CRESS is also a front-end for implementing features. It is possible to translate LOTOS and SDL to implementation languages such as C, thus realising features. However the CRESS tools also support more direct implementation through SIP CPL, SIP CGI and VoiceXML.

Industry seems to prefer graphical notations. Several graphical representations have been used to describe communications services. SDL (Specification and Description Language [9]) 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 services.[1] As a result, SDL service descriptions are not especially convenient or accessible to non-specialists. MSCs (Message Sequence Charts [8]) are higher-level and more straightforward in their representation of services. However neither SDL nor MSCs can readily describe the notion of features and feature composition.

Several notations have been specially devised for communications services. Among these, UCMs (Use Case Maps [2, 3]) and DFC (Distributed Feature Composition [19, 20]) are perhaps most similar in style to CRESS. Both lend themselves well to describing features and their composition, though the mechanisms are quite different from CRESS (plug-in maps for UCMs, pipe-and-filter composition for DFC). Both have been used successfully to model features and analyse their interactions. Unlike UCMs, CRESS allows both plug-in and triggered features. UCMs have been translated into LOTOS, but CRESS is explicitly designed to support translation into a number of target languages. DFC is primarily a software architecture, similar to work on ADLs (Architecture Description Languages [13]). However the means of feature composition has a natural graphical form.

The need for additional services was recognised early in SIP's development. SIP supports several mechanisms for user control of calls. CPL (Call Processing Language [11]) allows the user to manage call preferences, such as rejecting calls from certain addresses or forwarding calls based on caller and time of day. SIP also supports a web-like CGI (Common Gateway

[1] SDL does have something called a service, but this is not the usual kind of communications service.

Interface [12]) that is normally deployed in a server to intercept and act on SIP messages. A further solution is the SIP servlet, patterned after the idea of a Java servlet.

However, all of these are rather pragmatic. In the author's opinion, CPL is too high-level and (intentionally) too restricted to allow a full range of services to be created. On the other hand, SIP CGI is too low-level to allow services to be defined at an appropriate level of abstraction. Feature interaction in SIP has received limited attention, [10] being a notable exception. As will be seen, CRESS has been used to investigate services and features in SIP – their architecture, representation and analysis.

VoiceXML is aimed at IVR (Interactive Voice Response) systems. Being an application of XML, it is textual in form. However several commercial packages (e.g. Covigo Studio, Nuance V-Builder, Voxeo Designer) provide a graphical representation of VoiceXML. Some of these reflect the hierarchical structure of VoiceXML, while others emphasise the flow among VoiceXML elements. In the author's experience, these packages are (not surprisingly) very close to VoiceXML and do not give a sufficiently high-level view of VoiceXML services. More seriously, VoiceXML takes a pragmatic and programmatic approach. There is no way of formally analysing the correctness and consistency of a VoiceXML description. Interestingly, VoiceXML does not have the usual view of a feature (though it has roughly equivalent mechanisms). As will be seen, CRESS has been applied to VoiceXML services – graphical description, formalisation, feature composition, and analysis of services.

1.3 Overview of The Paper

The goal of this paper is to demonstrate that CRESS is a flexible notation of value in a number of domains. As background, Section 2 summarises the CRESS diagram format. Of necessity, the description is brief and condensed. Refer back to it when studying the diagrams that appear later. More on CRESS appears in [17, 18]. To complement previous CRESS work on IN services [17], Sections 3 and 4 show how CRESS can be applied to SIP and VoiceXML. It will be seen that SIP has affinities with IN telephony, but that VoiceXML supports very different kind of services. Nonetheless, the same notation can be used in all three domains for various purposes: representing services and their architecture, composing features, analysing features, and implementing features.

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. It follows that arcs between nodes should not be thought of as transitions between states. Arcs may be guarded by event conditions as well as value conditions. Perhaps most importantly, CRESS has explicit support for defining and composing features. CRESS has plug-in vocabularies that adapt it for different application domains. This allows CRESS diagrams to be thoroughly checked for syntactic and static semantic correctness. CRESS is also neutral with regard to target language (whether formal or programmatic), and can be translated into a number of languages.

2.1 Diagram Elements

A CRESS diagram is a directed, possibly cyclic graph. The oval nodes contain events and their parameters (e.g. *StartRing A B*). Events may also occur in parallel (|||). Events may be signals (input or output messages) or actions (like programming language statements). A **NoEvent** (or empty) node can be useful to connect other nodes. An event may be followed by assignments optionally separated by '/' (e.g. *Event / Busy A* $<-$ *True* sets *Busy(A)* to true). A node is identified by a number that may be followed by a symbol to indicate its kind, e.g.:

- '$<$' denotes an input node, while '$>$' denotes an output node (required only if the same signal can be received as well as sent)
- '+' starts a template that is appended to a matching node, while '–' indicates a template that is prefixed (used when describing features triggered by other behaviour).

The arcs between nodes may be labelled by guards. These may be either value expressions (imposing a condition on the behaviour) or event handlers (that are activated by dynamic occurrence of a condition). Event handlers are distinguished by their names (e.g. **NoInput**, triggered when the user does not respond to a VoiceXML prompt). If a graph is cyclic, it may not be possible to uniquely determine the initial node. In such a case, an explicit **Start** node is given; this is otherwise implicit. Comments may take several forms: text between parallel lines, hyperlinks to files, and audio commentary.

A CRESS diagram may contain a rule box (a rounded rectangle) that defines things like:

- the diagram variables and their types (e.g. **Uses Address** *A* **Value** *V*); temporary variables are predeclared for each type (e.g. addresses *A0..A9*, messages *M0..M9*, values *V0..V9*)
- the other services or features on which the diagram depends (e.g. **Uses** ... */ PROXY*)
- assignments triggered by signals (e.g. *Off-hook P / Busy P* $<-$ *True*, meaning that when phone *P* goes off-hook then it is noted as busy)
- macros (e.g. *Free P* $<- \sim$ *Busy P*, defining free as not busy for phone *P*)
- configuration information like the chosen features, translator options and user profiles.

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.

2.2 Service Architecture

CRESS diagrams usually rely on some infrastructure. For example, IN billing is handled by a separate subsystem with which call control cooperates. Similarly, call processing in the IN collaborates with the SCP (Service Control Point). It is therefore normal for CRESS to define a framework for each application domain. 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, user profiles, and feature-dependent data types.

A main CRESS diagram defines 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 may lead to nodes and guards being inserted, existing nodes and guard being replaced, and portions of the root diagram being deleted. This style of feature is appropriate for a one-off change to the original diagram. Suppose that a feature requires a PIN or password to be given before a call can proceed. This is a once-only action at the start of a call, and is appropriate for a spliced feature. The main disadvantage of this kind of feature is that it may have to duplicate large portions of the original diagram.

A template (macro) feature 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 realised statically by instantiating it using the parameters of the triggering event. The instantiated feature may be appended or prefixed to the triggering node. Since it is common for several features to be triggered by the same event, a number of feature instances may be chained. To help resolve certain categories of feature interaction at design time, CRESS supports priorities among features to control their order of application. For example after dialling, Abbreviated Dialling must be applied before Originating Call Screening. Some features are also cyclic, e.g. call forwarding may yield a new address that is itself subject to call forwarding. A loop back to the beginning of a feature is treated as a return to the start of the feature chain. This correctly handles billing, for example, if there are multiple call forwarding legs.

Although CRESS is mainly concerned with user services, it also supports ancillary aspects such as user profiles and billing. The services chosen by each user may be defined (e.g. call forwarding to a particular number, or call screening for particular callers). CRESS also supports billing. This might appear to be little more than logging the start and finish of calls. However, CRESS has explicit support for features like Charge Card, Freephone, Split Charging, and independent billing for each call leg.

2.3 Tool Support

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 9000 lines of code. Java would also have been a reasonable choice, but Perl was chosen because of its excellent support for systems programming. 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.

The CRESS toolset consists of five main tools, supported by seven underlying modules plus ancillary scripts. Internally the CRESS toolset comprises 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.

Figure 1 summarises CRESS application domains and target languages. For interactive voice services, CRESS uses VoiceXML as the implementation language. For SIP CGI, CRESS makes specialised use of Perl as the primary implementation language. (In general, a SIP CGI script may be almost any executable code.) Preliminary work has been undertaken on compiling into SIP CPL, but this is possible for only very limited forms of feature diagram.

	Target Language				
Domain	LOTOS	SDL	VXML	CGI	CPL
IN	✓	✓	✗	✗	✗
IVR	✓	✓	✓	✗	✗
SIP	✓	✓	✗	✓	✓✗

Figure 1: CRESS Language Support

3 SIP Services

3.1 Introduction to SIP

SIP [15] is an Internet standard for controlling sessions. In the context of Internet telephony, SIP is used to control voice calls. However SIP is a more general-purpose protocol that can be used to establish multimedia sessions such as video-conferences. SIP has also been adopted for use in call control for 3G (third generation mobile communication).

SIP is patterned after HTTP. The main requests are *Invite* (propose a session), *Cancel* (abort a request), *Bye* (close a session) and *Ack* (acknowledge the response to an *Invite*). Responses are identified by numeric code. CRESS identifies specific responses such as *Busy-Here*, *Ringing* and *Success*, as well as classes of response like *Failed* (error) and *Terminal* (unrecoverable error).

To establish a session, the caller sends an *Invite* to the callee. An *Invite* response such as *Success* elicits an *Ack* from the caller. To close a session, either party sends *Bye* and waits for the response. *Cancel* may be used to abort a previous request, mainly to cancel an *Invite* because a call attempt has been abandoned. Once a session is established, data flows directly between the users. That is, SIP is concerned only with session control and not session content.

Although superficially a straightforward protocol, SIP contains hidden complexity in its use of header fields. The SIP standard is also vague about unusual cases like cancelling an *Invite*, or crossover situations like receiving *Cancel*/*Bye* after sending *Cancel*/*Bye*. The standard has little formal definition of SIP, so its formalisation via CRESS is a useful clarification.[2]

3.2 CRESS Root Diagrams for SIP

Ideally, SIP services would be described purely from a user standpoint (i.e. without internal details of the protocol). This, for example, is how IN services are often formalised. A CRESS description of this external SIP behaviour *is* available as a single root diagram.

However it is in the nature of SIP services that they build on key events in the protocol (e.g. receiving an *Invite* or sending a *Bye*). It was also a goal of using CRESS to translate SIP service descriptions into CGI (Perl) and CPL. CRESS is therefore obliged to have some knowledge of protocol activities. A reasonable compromise has been reached by defining an abstract protocol interface. For example, this interface hides timeouts and the processing of header fields. Instead, the essential aspects of the protocol are made visible: requests and their main parameters (URIs, i.e. user addresses), responses and their codes. This abstract protocol interface is easily mapped onto the actual protocol. In OSI terms, CRESS maps user service

[2] Or, at least, it formalises what the author thinks SIP is meant to do!

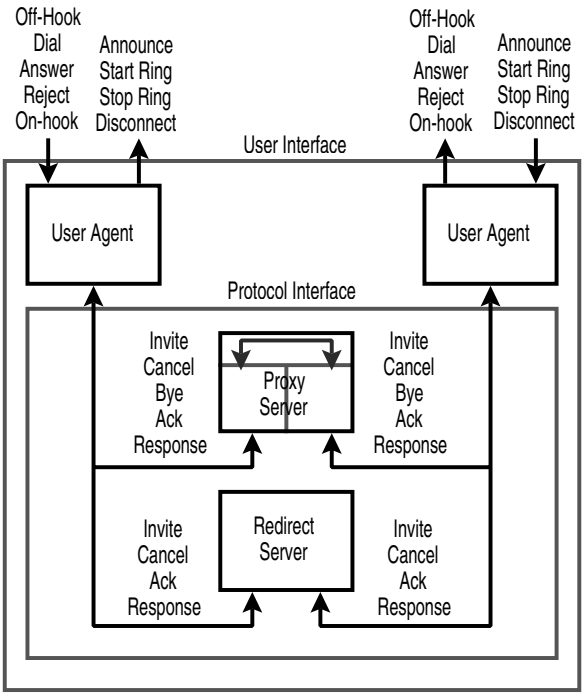

Figure 2: SIP Elements

primitives to the underlying SIP service primitives as shown in figure 2. For example, a user *Dial* request causes a SIP *Invite*.

SIP services can be deployed in three places: User Agents (which support the user interface to SIP), Proxy Servers (which relay and may manipulate requests), and Redirect Servers (which indicate how an *Invite* should be redirected to reach a user). As a consequence, the CRESS model of services exposes all three elements as shown in Figure 2. For familiarity, service primitives follow telephony terminology. Thus a user is said to go *Off-hook* or *On-hook*, though an actual phone may not be in use. Similarly a call results in *Start Ring* or *Stop Ring*, though this might be just a visual indication. A user may *Answer* or *Reject* an incoming call. *Announce* sends a call progress signal to the user. *Disconnect* means the other user has hung up.

The model of Figure 2 requires separate root diagrams for a User Agent, Proxy Server and Redirect Server; features modify each separately. For brevity, only a part of one root diagram is given in this paper. The terminating call side of a User Agent is shown in Figure 3; a separate diagram omitted here describes an originating call. The normal sequence of behaviour is as follows (with reference to node numbers in the figure). User *A* is the local callee, and user *B* is the remote caller. If an *Invite* is received by *A* from *B* (50), *A* starts ringing and *B* is notified of this (51). If *A* now answers (57), ringing stops and *B* is notified of successful setup (58). *B* acknowledges this with an *Ack* (59). Now both users can communicate. If *B* hangs up, a *Bye* is received (63) and *A* is notified of disconnection (64). *A* now hangs up (65) and *B* is informed of successful disconnection (66).

The rule box on the left defines how session status is maintained. For example, a user is noted as busy when going off-hook or as not busy when going on-hook. Observe that the notion of busy is *defined* and is not intrinsic to CRESS. For a conventional telephone or SIP phone, the definition of busy in Figure 3 is appropriate. However CRESS allows other definitions of busy. For example, a softphone is essentially never busy – a new call window

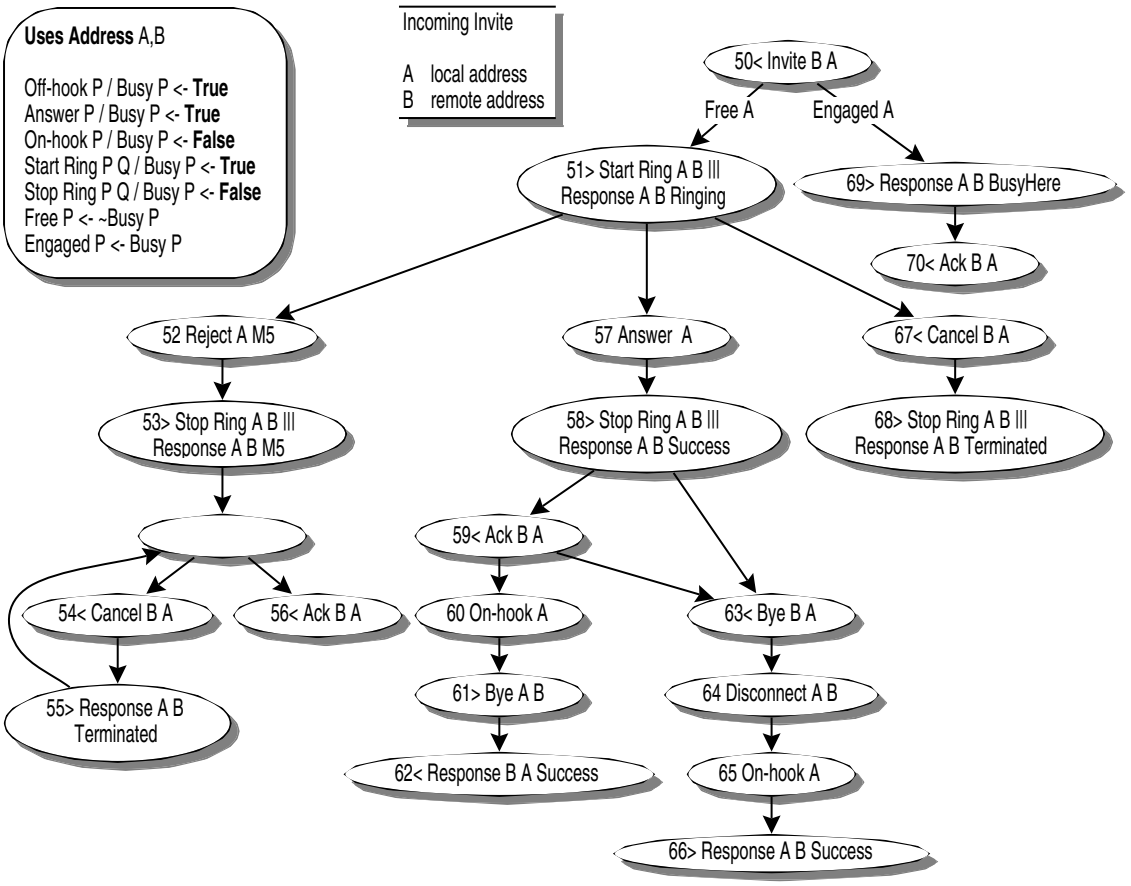

Figure 3: User Agent Root Diagram (Incoming *Invite*)

can be popped up at any time. A user may also be busy to certain callers (e.g. friends while at work) but free to others (e.g. managers). The time of the call might also influence whether the user is considered to be busy or not. Such factors can built into the definition of busy, or could be implemented as separate features. If appropriate, features may also define their own individual notion of busy.

3.3 CRESS Feature Diagrams for SIP

A SIP feature modifies the corresponding root diagram. Unfortunately, SIP features may vary in their definition according to where they are deployed. For example User Agents and Proxy Servers differ in their environment, and what they may initiate is also different.

Figure 4 shows a call forwarding template for a User Agent (left) and a Proxy Server (right). The '<' in template node 1 means it is triggered by input of an *Invite*, while the '+' means the template is appended to the matching node (e.g. figure 3 node 50). After substitution of *B* for parameter *P* and *A* for parameter *Q*, the template is copied and inserted in the root diagram. If the callee is busy but has a forwarding number (*ForwardBusy*), a User Agent reports that the callee has temporarily moved to this address (node 2). A Proxy Server, however, issues a new *Invite* (node 3) and handles the response (node 4).

Figure 5 shows that Terminating Call Screening is the same for a User Agent or a Proxy Server. If the caller is in the callee's screening list (*ScreenIn*), the call is declined (node 2).

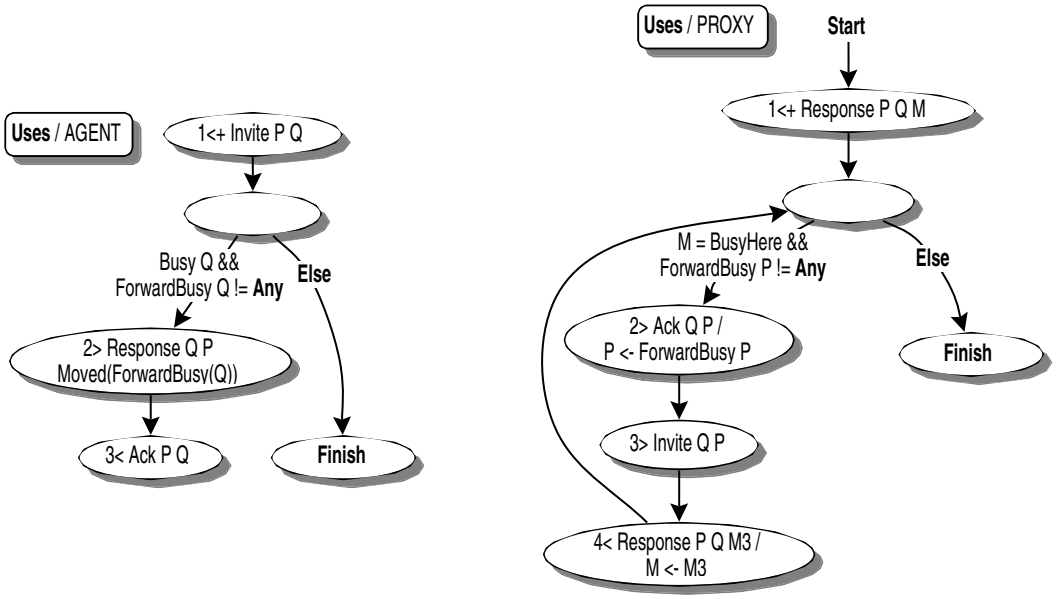

Figure 4: User Agent and Proxy Server Feature Diagrams (Call Forward Busy Line)

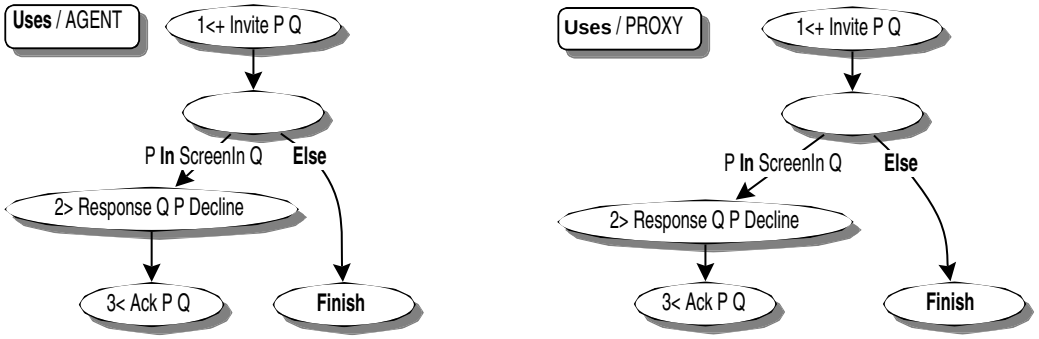

Figure 5: User Agent and Proxy Server Feature Diagrams (Terminating Call Screening)

3.4 CRESS as A Front-End for detecting SIP Interactions

When used for Internet telephony, SIP immediately lends itself to IN-like features. CRESS can readily be used to model SIP features such as Automatic Call-Back, Call Forwarding (several varieties), Call Screening (several varieties), Camp On Busy and Return Call.

As noted earlier, CRESS is a front-end for analysing and implementing features. Feature interactions are detected using separate techniques. For example, the author's approach in [16] is applicable to SIP. This considers an interaction to have occurred if a feature's behaviour changes in the presence of other features. Each feature is characterised by use-case scenarios derived from automatic simulation of the feature description in CRESS. The scenarios are represented as processes when using LOTOS or as MSCs when using SDL. The scenarios are not simply traces, but can include non-determinism, parallelism, and dependency on the presence of other features. A feature may be validated in isolation by running the scenarios on the feature combined with the corresponding root diagram. More usefully, a feature may be validated in combination with all other features. An interaction manifests itself as deadlock (because the feature cannot proceed as expected) or as non-determinism (because a triggering condition can lead to behaviour that is unexpected for the feature).

IN-like features such as the above can be readily represented using CRESS. Using the author's approach (or several others), it is easy to demonstrate feature interactions in SIP that are well known from the IN. For example, Call Forward Busy Line (Figure 4) interacts with Terminating Call Screening (Figure 5): trying to forward a call to a screened number will fail.

Certain kinds of IN interaction have different (or no) manifestations in SIP. As noted in [10], Internet telephony (including SIP) also introduces the possibility of new kinds of features and interactions. CRESS can discover the technical interactions in [10] (e.g. between Outgoing Call Screening and Call Forward). However, a number of the interactions in [10] concern user intentions. For example forking by a Proxy Server may lead to a call being picked up by voicemail, whereas the caller may prefer to wait for a person to answer. Such interactions are beyond the scope of CRESS (and indeed of most feature interaction approaches). The author is involved in separate work to tackle this [14]. The idea is to capture user intentions in the form of policies, and to perform resolution based on these.

Many features centre on busy, for example Automatic Call-Back, Call Forward Busy Line, Call Waiting and Camp On Busy. As noted already, busy may have a very different interpretation in SIP. As a result, busy-related features may not interact or may become irrelevant. Features related to call charging may also not apply. In a local or research environment, SIP calls are likely to be free. Features such as Charge Card, FreePhone and Split Charging are therefore irrelevant. However as SIP moves into a commercial phase, such features will become important. Calling Number Delivery is often a separate IN feature. However the address of the caller is in principle always available in SIP.

3.5 CRESS Translation of SIP Services

SIP CGI allows arbitrary features to be written. For example, a CGI script could query a database or invoke a complex algorithm. CRESS for SIP, however, describes features that perform only input, output, conditional tests and assignment. This is sufficient for many features but does not, of course, allow everything a CGI script might do. If it were necessary, SIP-specific actions could be included in CRESS much as VoiceXML-specific actions have been included. Whereas VoiceXML has a well-defined repertoire of actions that can be supported in CRESS, this is not feasible for SIP.

To give a feeling for how CRESS translates SIP services, the following LOTOS is an extract of what is generated for a User Agent incoming call (figure 3) as modified by Call Forward Busy Line (figure 4). After an *Invite*, the *Busy* and *ForwardBusy* values for *A* are determined. If *A* is busy but has a forwarding address, a *Moved* response is sent with this forwarding address. Following the *Ack* from *B*, the session instance ends. As seen below, the CRESS translator automatically adds comments so the LOTOS can be related directly back to diagrams. The diagram label *AGENT_CFBL_1* refers to the User Agent's Call Forward template, instance 1.

```
Recv !Invite ?B:Address ?A:Address;                          (* AGENT input 50 *)
Stat !Read !Busy !A;                                             (* read status *)
Stat !Read ?Busy_A:Bool;
Stat !Read !ForwardBusy !A;                                      (* read status *)
Stat !Read ?ForwardBusy_A:Address;
(
  [Busy_A And (ForwardBusy_A Ne AnyAddress)] >>                  (* condition valid? *)
    Stat !Read !ForwardBusy !A;                                  (* read status *)
    Stat !Read ?ForwardBusy_A:Address;
```

```
        Send !Response !A !B !Moved(ForwardBusy_A);          (* AGENT_CFBL_1 output 2 *)
        Recv !Ack !B !A;                                     (* AGENT_CFBL_1 input 3 *)
        Stop                                                 (* end of behaviour *)
  []
    [Not (Busy_A And (ForwardBusy_A Ne AnyAddress))] ->      (* condition invalid *)
        ...
)
```

4 VoiceXML

4.1 Introduction to VoiceXML

VoiceXML [5] derives from earlier work on scripting languages for interactive voice services. VoiceXML is designed to support what users wish to do in a call – talk, as opposed to choosing selections by using a keypad. VoiceXML is receiving impetus from widespread use of mobile telephony (where a user on the move may not have web access). The need for access by the partially sighted or disabled is also a strong motivation for voice services.

VoiceXML is a mixture of the declarative and the procedural, the event-driven and the sequential. The underlying model is that the user 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, using speech recognition to extract digital data. Some actions may be governed by a condition or a count that specifies when the action is permitted. For example a different prompt may be given on the third attempt at input, or a field may be selected only when some condition is true. VoiceXML also supports a hierarchical event model. A script may throw an event, aborting the current behaviour and activating a matching event handler.

The goal of using CRESS with VoiceXML is to define the key aspects of an interactive voice service. The advantages of CRESS over using VoiceXML directly are:

- Services are represented at a more abstract level. VoiceXML is very close to the realisation of a service. As a result, it is easier to grasp the essence of a service described in CRESS.
- There is no formal definition of VoiceXML. Indeed, some concepts in VoiceXML are only vaguely described (e.g. the meaning of events) and some are defined loosely (e.g. the semantics of expressions and variables). CRESS thus contributes to a more precise understanding of VoiceXML.
- A large VoiceXML script typically consists of many documents with many parts. It can be difficult to check whether the script is self-consistent, e.g. will not loop or end prematurely. As far as the author can tell, VoiceXML in practice is developed by manual debugging. CRESS gives the immediate benefit of translation to a formal language (LOTOS, SDL). The resulting specification can be rigorously analysed, e.g. automated techniques can be used to detect unspecified receptions, unreachable states, deadlocks and livelocks.

Speech synthesis markup can be included in a prompt, e.g. for emphasis or to spell out a word. The markup is preserved on translation to VoiceXML, but ignored on translation to LOTOS or SDL. Variable values may also be interpolated, using *$variable* to say the value of this variable. As a special case, *$enumerate* is used to speak the options of the current field. VoiceXML can also interpolate variable values, but the syntax is more awkward than in CRESS.

In practice, VoiceXML applications are often written as a number of documents containing a number of forms. This is the most natural form of modularity in VoiceXML. However

CRESS	VoiceXML	Interpretation
Audio *message*	<**audio**> *message*	Speak message
Clear *variables*	<**clear**> with namelist *variables*	Reset prompt counter, undefine variables
Option *variable prompt options*	<**field**> name *variable*, <**prompt**>, <**option**>s	Start new field, prompt for input, analyse input using options, set variable from input
Prompt *message*	<**prompt**> *message*	Speak prompt
Reprompt	<**reprompt**>	Re-process current form, usually causing most recent prompt to be re-issued
Request *variable prompt grammar*	<**field**> name *variable*, type *grammar*, <**prompt**>	Start new field, prompt for input, analyse input using grammar, set variable from input
Retry	Undefine current <**field**> variable, <**reprompt**>	Restart current form, re-inputting most recent field
Submit *URI variables*	<**submit**> to *URI* the namelist *variables*	Send values to web server URI (usually an executable script)

Figure 6: CRESS-VoiceXML Correspondence

a VoiceXML application can be considered as a single document with a single form, and this is how it is represented in CRESS. The fields of a form can be mimicked 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. For this reason, fields are deliberately not prominent in CRESS and are instead introduced implicitly.

For those unfamiliar with VoiceXML, Figure 6 outlines its main capabilities. For those familiar with VoiceXML, this figure indicates the correspondence with CRESS. In a number of cases, an optional condition or count may be given after the action. For example, a different prompt can be issued for the fourth attempt[3] at entering a field: **Prompt** ″State your name″ 4.

Actions appear in CRESS diagram nodes. Plain arcs between nodes are used for sequences of actions. Arcs may also be qualified by guards that are value expressions or event handlers. Events include **Error** (run-time script error), **Exit** (script exited), **NoInput** (no user response to prompt) and **NoMatch** (user response did not match expected form of input). These are all shorthands for the more general form of **Catch** plus an event list. **Filled** acts like an event handler, though it is not treated as such in VoiceXML. If the user responds appropriately to a prompt, the input is stored in the field variable and the **Filled** ‘event’ occurs.

4.2 CRESS Root Diagrams for VoiceXML

CRESS is not a direct graphical representation of VoiceXML. This would be pointless since most commercial tools for VoiceXML do this anyway. In fact the structuring mechanisms of CRESS are completely different from those of VoiceXML. Both support actions, sequences and alternatives. The flow of control in CRESS is quite visible; in VoiceXML it can be hard to determine, because it 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 structures such as loops in a diagram. These have to be coded indirectly in VoiceXML, so

[3]Actually the prompt is issued if the count is $>= 4$, but less than the next highest prompt count for that field.

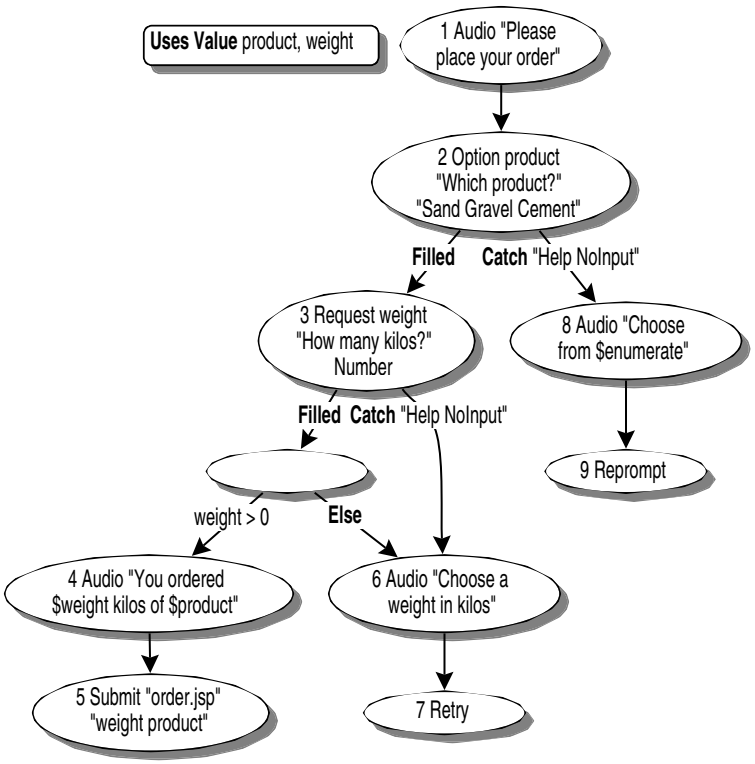

Figure 7: VoiceXML Root Diagram for Quarry Ordering Application

the CRESS structure is clearer. Although VoiceXML supports a **GoTo** which appears to be equivalent, this can be used only for transitions to another field or document. As a result, it is not obvious how to translate a directed cyclic graph like a CRESS diagram into VoiceXML.

CRESS expects to have a definition of root behaviour. There appears to be something similar in VoiceXML – an application root document. However this is very restrictive, and may contain only variables, event handlers and elementary definitions that are common to the documents of a VoiceXML application. As a result, a CRESS root diagram is taken to be the core behaviour of a VoiceXML application. This is not unique, in the way that POTS is the obvious choice for the IN or a User Agent for SIP. Every VoiceXML application therefore has its own root diagram.

For concreteness, Figure 7 shows a root diagram for a VoiceXML application. It is supposed that the hypothetical Quarry Inc. requires interactive telephone ordering of its products. The description invites the caller to order a *product* (sand, gravel, cement) and the required *weight*. These items are then submitted to the *order.jsp* Java servlet. If the user asks for help (simply by saying 'Help') or says nothing, an explanation is given and the user is re-prompted. In the case of *weight*, the user is reprompted if the value is not positive. **Retry** is used to clear the value entered for *weight*, otherwise the field will be ignored on the reprompt because it has already been filled. (This is how VoiceXML behaves. The programmer must force a field to be re-entered if it has already been filled.)

4.3 CRESS Feature Diagrams for VoiceXML

VoiceXML lacks the telephony concept of feature as a behaviour that may be triggered by some condition. The nearest equivalent in VoiceXML is a subdialogue (resembling a subrou-

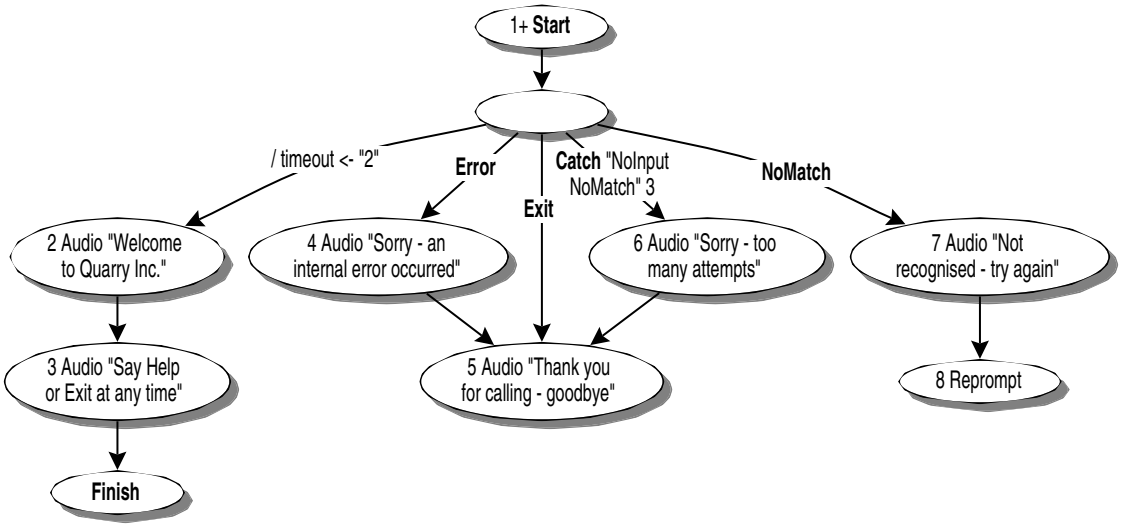

Figure 8: VoiceXML Feature Diagram to introduce Quarry Inc. Applications

tine). A subdialogue may have parameters and return results. Subdialogues are executed in an independent interpreter context, making it difficult to share certain information (such as prompt count). This limits the value of subdialogues as features. In VoiceXML, the best that can be done is to *explicitly* invoke some code as a 'feature'. This means that any such 'feature' needs to be written into the original code. It is not possible to invoke features automatically in the way that is common with the IN (or SIP). A VoiceXML programmer would probably regard this a good thing since there are no hidden surprises. Triggered features have, however, proven their worth in telephony. They are therefore supported by CRESS even though the concept of feature is unknown to VoiceXML.

As examples of desirable features, suppose that Quarry Inc. has a range of applications besides the ordering application in Figure 7. There might, for example, be separate applications to modify an order, pay an account, or change the delivery address. It would be desirable to ensure a consistent VoiceXML treatment of these applications. For example, there should be the same default handling of events and a common introduction to the applications. It would also be worthwhile to request confirmation before anything is submitted to a web server.

Figure 8 defines an introductory environment to modify any root diagram. Common handlers are defined for various events. The feature is placed just after the **Start** node in the root diagram (implicit before Figure 7 node 1). In the absence of event handlers like those in Figure 8, platform-defined handlers are used that may not be suitable in general. Although an application is likely to deal with **NoInput** and **NoMatch** on a per-field basis, figure 8 ensures that after three such failures the user is disconnected. Figure 8 shows that general VoiceXML properties can be defined; here the timeout for no input is set to two seconds (*timeout* $<-$ *2*). Welcome messages are also spoken before executing application-specific code.

Figure 9 defines a confirmation feature that will ask the user to proceed before submitting information to a web server. This feature is triggered by a **Submit** action, but executed before it (as indicated by the '–' in the first template node number). If the user says 'yes', execution continues with submission. Otherwise, the current fields are cleared and the user is reprompted.

Even small VoiceXML features can be useful. Figure 10 shows one that inhibits input timeout (value 0), and one that prevents prompts from being interrupted (so-called barge-in).

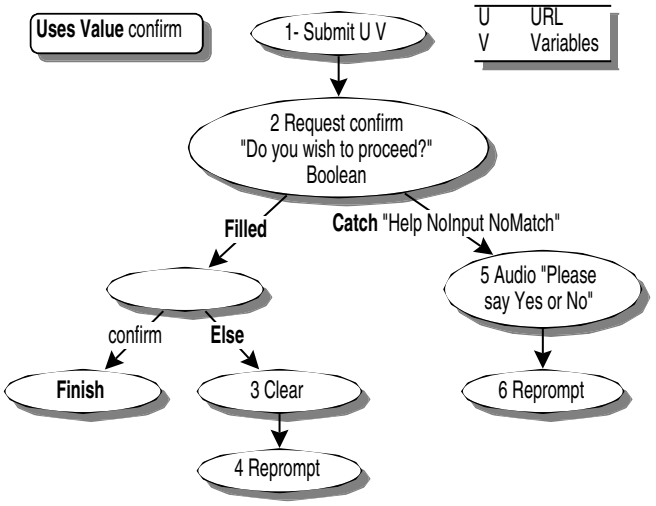

Figure 9: VoiceXML Feature Diagram for Confirmation

Figure 10: VoiceXML Feature Diagrams for No Timeout and No Interruption

4.4 CRESS as A Front-End for detecting VoiceXML Interactions

As for SIP, CRESS is merely a way of representing services and features. Separate detection techniques must be used on the chosen formalism for the diagrams. In point of fact, CRESS is perhaps most useful for checking the integrity of a VoiceXML description (freedom from deadlock, etc.). Just as features are foreign to VoiceXML, so is feature interaction. However several general categories of 'feature interactions' can be identified:

- Platform properties may be defined hierarchically. For example, the generic timeout in Figure 8 may be overridden within a field by some feature. From the user's point of view, this could lead to a small but observable change in behaviour.
- Two features may change an application variable inconsistently, leading to an interaction.
- Event handlers are defined in a hierarchy. When an event occurs, the VoiceXML interpreter looks upwards in the hierarchy for the appropriate handler. For example, consider Figures 7 and 8. If there is no input in response to the *product* prompt, execution follows the field handler (figure 7 node 8). However after three failures to input, the generic handler will be invoked (figure 8 node 6). A consequence of this is that features may unexpectedly override the usual handling of an event. From the user's point of view, a certain combination of features could result in different behaviour.
- A more subtle interaction can arise if several input grammars are active at once. User input may therefore be parsed in a different way if certain features are combined.
- Another indirect interaction arises with use of DTMF (Dual-Tone Multi-Frequency) responses. 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 will be completely altered.

All these cases lead to unexpected changes in behaviour when certain features are present, so feature interaction is detected as normal in CRESS. VoiceXML may also be involved indirectly in conventional feature interactions since scripts are allowed to initiate phone calls. These may suffer from the usual telephony interactions, for example call screening might interfere with call forwarding. Since such interactions are external to VoiceXML and therefore to CRESS, the VoiceXML diagrams themselves will not help to find the problems.

4.5 CRESS Translation of VoiceXML Services

VoiceXML is a large language embedded in an even larger framework. For example, VoiceXML includes support for ECMASCRIPT (JavaScript). It also supports complex grammars for speech recognition and markup for speech synthesis. VoiceXML is integrated with other technologies such as database access and web servers. It is not feasible to represent the entirety of such voice-based services. Instead, CRESS concentrates on the essential aspects of VoiceXML control. Limited support is provided for ECMASCRIPT (specifically for numerical, string and logical expressions). External aspects such as databases and the web are outside CRESS.

CRESS focuses on VoiceXML control. Special parameters relevant only to a VoiceXML interpreter can be given in a diagram at the end of an action. They are copied literally when CRESS is converted to VoiceXML, but are ignored for translation to other target languages. For example, a diagram usually just contains audio prompt such as **Audio** ″State your name″. However optional VoiceXML parameters may also be given, such as the URI of a source sound file and a timeout (2 sec.) for fetching this:

Audio ″State your name″ src=′′http://www.server.org/name.wav″ fetchtimeout=″2″

VoiceXML allows application-specific speech grammars to be defined. It is not practicable to translate these into, say, LOTOS or SDL. Instead, CRESS supports only standard VoiceXML grammars such as *boolean*, *number* and *time*. A CRESS specification framework includes the ability to parse such inputs.

To give a feeling for how CRESS translates IVR services into VoiceXML, the following corresponds part of the quarry ordering application (figure 7) where it is modified by the confirmation feature (figure 9). The CRESS **Request** becomes a VoiceXML field that fills in the *confirm* value. If this is assigned *true*, the order values are submitted to the server. Otherwise the form that invoked confirmation is cleared and the user is prompted for new values. If the user asks for help, does not say anything or says something invalid, an event handler catches this and reprompts the user. The diagram label *CONFIRM.1* means the *Confirm* template, instance 1.

```
<field name='confirm' type='boolean'>          <!-- CONFIRM.1 node 2 field 'confirm' -->
  <prompt>                                     <!-- CONFIRM.1 node 2 prompt -->
    'Do you wish to proceed?'
  </prompt>
  <catch event='help noinput nomatch'>                          <!-- catch event -->
    <audio>                                    <!-- CONFIRM.1 node 5 audio -->
      'Please say Yes or No'
    </audio>
    <reprompt/>                                <!-- CONFIRM.1 node 6 to form top -->
  </catch>                                                      <!-- end catch -->
  <filled>                                                      <!-- filled event -->
    <if cond='confirm'>                                         <!-- check confirm -->
```

```
      <submit expr='order.jsp' namelist='weight product'/>   <!-- ORDER node 5 to server -->
      <exit/>                                                <!-- exit script -->
    <else/>                                                  <!-- else after confirm -->
      <clear/>                                               <!-- CONFIRM.1 node 3 clear -->
      <reprompt/>                                            <!-- CONFIRM.1 node 4 to form top -->
    </if>                                                    <!-- else after confirm -->
  </filled>                                                  <!-- end filled -->
</field>                                                     <!-- end field -->
```

5 Conclusions

It has been shown that CRESS is a flexible notation that can describe a variety of voice services and features – the IN in previous work, and now SIP and VoiceXML. SIP lends itself to a telephony treatment, so many IN-like features can be described and many IN-like interactions can be detected. As has been seen, VoiceXML is rather different in character. Nonetheless VoiceXML services can usefully be described in CRESS, and a meaningful interpretation can be given of features in this context.

In all cases, CRESS is the front-end that describes services/features, composes them, and translates them to a target languages for analysis or implementation. CRESS thus separates representation from analysis, and supports a variety of specification languages. CRESS complements existing interaction detection techniques, enabling them to be applied in new areas.

The plug-in architecture of CRESS has now been demonstrated in three different domains. 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 services for WSDL (Web Services Description Language).

Acknowledgements

The author is grateful to Stephan Reiff-Marganiec (University of Stirling) for discussions on VoiceXML, and for reviewing a draft of this paper.

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, R. J. A. Buhr, T. Gray, and L. M. S. Logrippo. Use case maps for the capture and validation of distributed systems requirements. In *Proc. 4th. IEEE International Symposium on Requirements Engineering*, pages 44–53. Institution of Electrical and Electronic Engineers Press, New York, USA, June 1999.

[3] 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.

[4] M. H. Calder, E. H. Magill, and D. J. Marples. A hybrid approach to software interworking problems: Managing interactions between legacy and evolving telecommunications software. *IEE Software*, 146(3):167–180, June 1999.

[5] V. Forum. *Voice eXtensible Markup Language*. VoiceXML Version 1.0. VoiceXML Forum, Mar. 2000.

[6] Q. Fu, P. Harnois, L. M. S. Logrippo, and J. Sincennes. Feature interaction detection: A LOTOS-based approach. *Computer Networks*, 32(4):433–448, Apr. 2000.

[7] 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.

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

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

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

[11] J. Lennox and H. Schulzrinne, editors. *CPL: A Language for User Control of Internet Telephony Services*. Internet Draft CPL-05. The Internet Society, New York, USA, Nov. 2001.

[12] J. Lennox, H. Schulzrinne, and J. Rosenberg, editors. *Common Gateway Interface for SIP*. RFC 3050. The Internet Society, New York, USA, Jan. 2001.

[13] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In *Proc. 6th. European Software Engineering Conference/Proc. 5th. Symposium on the Foundations of Software Engineering*, pages 60–76, Zurich, Switzerland, Sept. 1997.

[14] S. Reiff-Marganiec and K. J. Turner. Use of logic to describe enhanced communications services. 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 130–145. Springer-Verlag, Berlin, Germany, Nov. 2002.

[15] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnson, J. Peterson, R. Sparks, M. Handley, and E. Schooler, editors. *SIP: Session Initiation Protocol*. RFC 3261. The Internet Society, New York, USA, June 2002.

[16] 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.

[17] 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.

[18] 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.

[19] P. Zave. Architectural solutions to feature-interaction problems in telecommunications. In K. Kimbler and W. Bouma, editors, *Proc. 5th. Feature Interactions in Telecommunications and Software Systems*, pages 10–22. IOS Press, Amsterdam, Netherlands, Sept. 1998.

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