Kenneth J. Turner. Formalising Graphical Service Descriptions using SDL. In Rick Reed and Jeanne Reed, editors, Proc. SDL 2003, LNCS 2708, pages 183-202, copyright Springer-Verlag, Berlin, Germany, July 2003.

Formalising Graphical Service Descriptions using SDL

Kenneth J. Turner

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

Abstract. It is convenient to describe telecomms services using a graphical notation that is accessible to non-specialists. However, the notation should also have a formal interpretation for rigorous analysis. CRESS (Chisel Representation Employing Systematic Specification) has been developed for this purpose. A brief overview of CRESS is given. It is explained how features (additional services) can be defined in a modular fashion, and automatically combined with a base service. Brief case studies illustrate how the approach has been used to describe services in the IN (Intelligent Network), SIP (Session Initiation Protocol), and IVR (Interactive Voice Response). Finally, it is shown how CRESS diagrams are translated into SDL for automated simulation, validation and implementation.

1 Introduction

A telecomms service is a set of capabilities packaged and sold to end-users, while a telecomms feature is a self-contained aspect of a service. However, the terms 'service' and 'feature' tend to be used interchangeably. It is costly and time-consuming to develop a new telecomms service. It is therefore desirable to have a precise description of what is to be built. A critical lesson from telephony is that services often interfere with each other in unexpected and undesirable ways – so-called feature interaction [2].

A formal description can be used for automated analysis of service incompatibilities. However, both technical and non-technical people must cooperate in defining a service. A formal description is likely to be understood only by specially trained engineers. It is therefore desirable that service descriptions be meaningful to the non-specialist, such as a manager or a marketing person. The ideal service description should be graphical (accessible to non-technical staff), abstract (permitting a variety of implementations), and precise (automatically translated into a formal language)

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.

The author's approach to defining and analysing services is a graphical notation called CRESS (Chisel Representation Employing Systematic Specification). CRESS is a significant extension of the original Chisel notation developed by BellCore [1]. The author was attracted by the simplicity, graphical form, and industrial orientation of Chisel. Although it has mainly been used in telecomms, CRESS is not tied to this. Indeed, CRESS supports plug-in application domains that define the vocabulary used to

describe services. Applying CRESS to a new application mainly requires the definition of a new vocabulary for events, types and system variables.

CRESS is neutral with respect to the target language. It can, for example, be compiled into SDL (Specification and Description Language [8]) and LOTOS (Language Of Temporal Ordering Specification [5]). This gives formal meaning to services defined in CRESS, and allows rigorous analysis of services. The semantics of a CRESS diagram is thus defined by the target language. In principle, this risks inconsistency. In practice, CRESS diagrams are sufficiently simple that they can be given the same interpretation in different languages.

For direct implementation in certain domains, 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.

This paper explains how SDL is used to support CRESS. The general approach is illustrated with examples taken from telephony with the IN (Intelligent Network [6]), Internet telephony with SIP (Session Initiation Protocol [10]), and IVR (Interactive Voice Response) using VoiceXML (Voice Extended Markup Language [3]).

Several graphical representations are used to describe communications services. SDL is the main formal language used in telecomms. Although it has a graphical form, SDL is a general-purpose language that was not especially designed to represent services. SDL service descriptions are not accessible to non-specialists as they tend to be low-level and rather technical. MSCs (Message Sequence Charts [7]) are higher-level and more straightforward in their representation of services. However neither SDL nor MSCs can readily describe the notions of feature and feature composition that are important in defining services. CRESS was designed to fill this gap.

Since CRESS can be translated to SDL, then of course the same services could be described directly in SDL. This is not done for a variety of reasons:

- The CRESS diagrams are much more compact than their SDL equivalent. CRESS diagrams also require less technical knowledge than SDL. As shown by BellCore [1], even non-technical people can understand diagrams in the style of CRESS.
- CRESS service descriptions are more abstract than their equivalent in SDL. Although UML could be used, services do not usually lend themselves to an object-oriented treatment. (This is a reflection on typical services, not UML.)
- CRESS has explicit support for features and feature composition. Only approximations to these exist in UML or SDL. As a result, it would be necessary to re-invent this support – which is what CRESS does natively.
- CRESS is *language-independent*. The same diagrams can be translated to different target languages. This is very powerful, because the same diagrams can then be used for different purposes. Consider, for example, an IVR service. This can be formulated in CRESS for discussion among technical and non-technical staff. It can then be translated into SDL, where reachability analysis and MSC-based validation are very convenient. It can also be translated into LOTOS, where model-checking, theorem-proving and test generation algorithms are available. Finally, it can be translated into VoiceXML for actual implementation.

– CRESS provides domain-specific frameworks. For example, the IN framework has built-in support for SCPs (Service Control Points), user profiles and billing. These would all need to be devised and specified using a plain SDL approach.

The paper aims to show that CRESS, supported by SDL, can be used with a variety of kinds of services. Section 2 provides a brief overview of the CRESS diagrammatic notation. It is not feasible to provide a tutorial on CRESS here, but more information can be found in [11, 12]. Sections 3, 4 and 5 show how CRESS can be applied to services in the IN, SIP and VoiceXML; some knowledge of these areas is assumed. Finally, section 6 discusses how CRESS diagrams are turned into SDL and then analysed.

2 The CRESS Notation

This section gives a compact overview of the CRESS notation used in this paper. For concreteness, examples are taken from diagrams that appear later in the paper.

2.1 Diagram Elements

A CRESS diagram is a directed, possibly cyclic graph. 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). An event may be followed by assignments optionally separated by '/'. A ***No-Event*** (or empty) node, meaning no event occurs, can be useful as a connector. It may join a number of preceding and following nodes as a more compact way of linking all the nodes.

Nodes are identified by a number, which may be followed by a symbol to indicate the kind of node. For example, '$<$' denotes an input node, while '$>$' denotes an output node. A feature start node is marked '+' or '$-$', depending on whether it is appended or prefixed to the matching node that triggers it. Sometimes it is necessary to prevent a node from matching a feature template by appending '!' to its node number.

As an example of a node, the following appears in figure 6:

2> Ack Q P / P <− ForwardBusy P

This output node, numbered 2, sends a SIP acknowledgement from caller *Q* to callee *P*. The callee address *P* is then replaced by the one used for forwarding on busy. Parentheses are often omitted in CRESS if there is no ambiguity, so *Ack Q P* for example means *Ack(Q,P)* and *ForwardBusy P* means *ForwardBusy(P)*.

A directed arc between nodes links them in sequence. Branching is permitted if there is a choice of events. The arcs between nodes may be labelled by guards. These are either value expressions (imposing a condition on the behaviour) or event triggers (that are activated by dynamic occurrence of a condition). Event triggers are distinguished by their names. Examples of guards appear in figures 2 and 8:

Free A B in the IN means that *B* is free for a call from *A*

Filled in VoiceXML triggers the behaviour for correct user input.

A CRESS diagram can contain a rule box (a rounded rectangle) that defines things such as the diagram variables, parent diagrams, macros, and configuration information like subscriber profiles. Sample rule boxes appear in figures 2 and 3:

Uses Address *A,B* defines diagram variables *A* and *B* of type ***Address***

Uses */ POTS* says the parent diagram is that for *POTS*

Off-hook P / Busy P <- ***True*** is triggered by signal *Off-hook* with parameter *P*; it notes the status of phone *P* as busy when it goes off-hook.

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 copy-and-paste or by triggering.

2.2 Services and Features

CRESS diagrams are interpreted in the context of a specification framework that defines a domain-specific infrastructure. For example, IN billing is handled by a separate subsystem that cooperates with call control. Similarly, call processing in the IN collaborates with an 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 are handled by the CRESS preprocessor. For SDL, the macro call *Cress(Types)* generates ***Signal***, ***SignalList*** and ***Type*** definitions appropriate to the application domain. This information is determined by the plug-in vocabulary. Feature diagrams are automatically combined with the root diagram by the macro call *Cress(Features)*. If there are several root diagrams, each root and its features are combined by a macro call such as *Cress(Features/Proxy)* for a SIP Proxy Server. Finally, the macro call *Cress(Profiles)* automatically generates configuration-specific details and the subscriber profiles. This information is defined by a special CRESS configuration diagram.

A main CRESS diagram defines the root (or basic) behaviour. Although this may be the only diagram, CRESS also supports feature diagrams that modify the root diagram (or other features), augmenting basic behaviour with new or modified capabilities.

A spliced (plug-in) feature adds to a root diagram by copy-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.

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 at a ***Finish*** (or empty) node. At this point, behaviour resumes from the triggering node in the original diagram. A template feature is realised statically, instantiating it with the parameters of the triggering event. The instantiated feature may be appended or

prefixed to the triggering node. Since it is common for features to be triggered by the same event, feature activations may be chained. In such a situation, CRESS defines priorities for features to control their order of application. Some features are cyclic, e.g. call forwarding may yield a new address that is itself subject to call forwarding (see figure 6). A loop back to the beginning of a feature is treated as a return to the start of the feature chain.

Although CRESS is mainly concerned with user services, it also supports ancillary aspects such as subscriber profiles and billing (dependent on the application domain). Profiles define the services chosen by each subscriber. CRESS also has explicit support for billing features such as credit card calling 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 13,000 lines of code. Java would also have been a possibility, but Perl is just as portable and is very convenient for CRESS purposes. A traditional compiler deals with textual languages. However CRESS is a graphical language, and this creates interesting challenges such as compiling cyclic rather than hierarchical constructs.

The CRESS toolset consists of five main tools. Including test scenarios, there are about 600 supporting files for all domains and target languages. Internally the CRESS toolset 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 (including one for SDL). CRESS has been designed to work with a number of diagram formats. Currently, diagrams are drawn with the *Diagram!* tool that runs on five different platforms. However a platform-independent diagram editor is currently being implemented using the open-source *jGraphPad*.

CRESS supports the IN, SIP and VoiceXML domains. For formal analysis, SDL and LOTOS are the target languages. For SIP, the primary target language is a specialised use of Perl for CGI scripts. Preliminary work has been undertaken on compiling into SIP CPL, but this is possible for only very limited forms of feature diagram. For interactive voice services, VoiceXML is the obvious target language for implementation.

The CRESS user draws root and feature diagrams using an appropriate graphical editor. A single button-click retrieves all the diagrams, combines them with the appropriate specification framework, and translates them to SDL. Validation and verification may use any standard SDL mechanism such as reachability analysis. However it is often most convenient to use MSC validation. Each feature is characterised by a set of use-case scenarios expressed as MSCs. These may be created manually, or may be derived from simulation of the feature. Validating a feature then means validating these MSCs. Features can be validated in isolation. This is useful to gain confidence in the feature description. More usefully, features can be validated in combination – the most complete test being when all features are deployed simultaneously.

In fact, the CRESS design procedure already eliminates a number of feature interactions. For example, CRESS requires the specifier to prioritise features so that they are applied in an appropriate order. A common interpretation of feature interaction is that a feature operates correctly on its own but not in combination with other features.

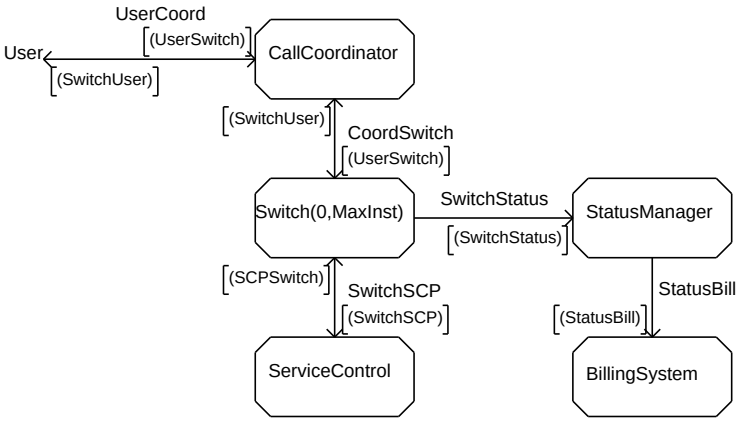

Fig. 1. CRESS SDL Structure for the IN

Feature interaction shows up as failure to validate an MSC. Once characteristic MSCs have been defined, the validation process is completely automated. This is particularly valuable when new features are added to a system. The existing MSCs can be used as regression tests to check that existing features still behave correctly.

3 Case Study: IN Services

The IN (Intelligent Network [6]) defines an architecture for flexible support of services in telephony. A key contribution is the separation of call switching in SSPs (Service Switching points) from service handling in SCPs (Service Control Points). The IN has been used to implement a wide variety of services such as various forms of billing (e.g. split charging), busy handling (e.g. call waiting), call forwarding (e.g. on no answer), call screening (e.g. on caller identification), and conferencing (e.g. three-way calls).

The goals of using CRESS with the IN are:

- to define features in a comprehensible way
- to allow features to be validated in isolation
- to validate features in combination, checking for feature interactions.

The CRESS specification framework for the IN occupies 15 pages of SDL/GR, so only the top level can be given here; more detail can be found in [11]. The main block structure is shown in figure 1. Telephone users communicate with the network via the *User* channel. This allows users to send signals such as *Off-hook*, *Dial* and *On-hook*. The network can send the user signals such as *DialTone*, *Disconnect* and *StartRinging*. The internal structure of the network is hidden from the users. It comprises the following processes whose behaviour is fixed, except for that of *Switch*:

Switch is multiply instantiated up to the concurrent call limit *MaxInst*. It describes the operation of a switch (SSP) provisioned with features. The switch behaviour is that of the POTS root service defined in CRESS, modified by the features described in separate CRESS diagrams.

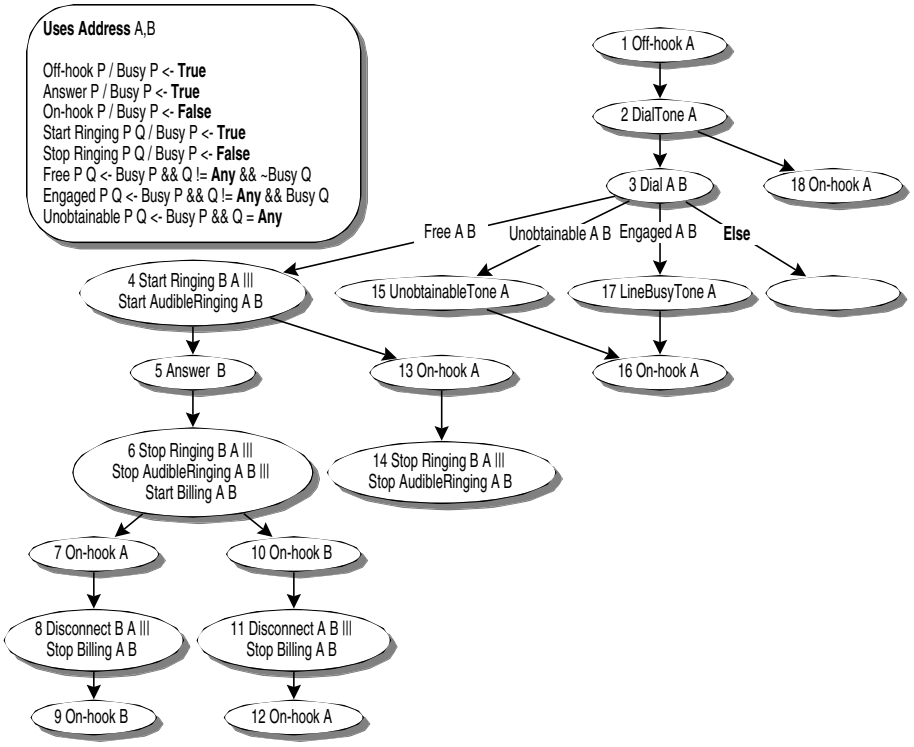

Fig. 2. CRESS Root Diagram for POTS

CallCoordinator is responsible for linking users and their calls. For example if a user goes off-hook, it is necessary to create an instance of the *Switch* process and send it the *Off-hook* signal. The *CallCoordinator* process therefore acts as a distributor of user signals.

StatusManager maintains the status and profiles of all users. The dynamic status of each user is maintained, e.g. whether busy or the number of the last caller. The profiles define the user's telephone number, which features have been selected, and the parameters of these features (e.g. a forwarding address). Subscriber profiles are provided by the CRESS configuration diagram.

BillingSystem logs all billing information. Since SDL does not directly support external files, the raw billing data is merely absorbed by this process (i.e. caller and callee, paying party, start and finish times of call). However this information can be used in (say) simulation or validation output.

ServiceControl describes the operation of an SCP. It provides the support expected of an IN SCP. It also retrieves user statuses and profiles from the *StatusManager*.

Figure 2 shows the CRESS root diagram for POTS. This (plus features) is used to generate the body of the *Switch* process in figure 1. The IN model deals with complete calls (i.e. caller plus callee behaviour) and not just one side of a call. CRESS diagrams

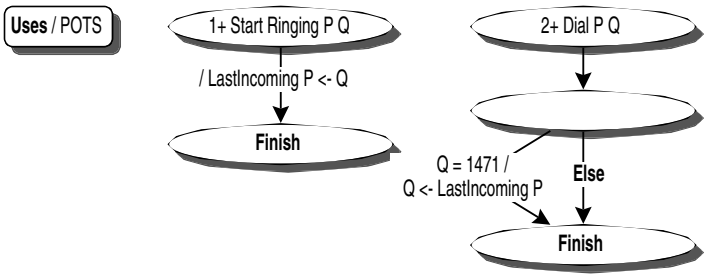

Fig. 3. CRESS Feature Diagram for Return Call

have also been created for 11 typical IN features, based on descriptions in the first feature interaction contest [4]. Most features are switch-based, but those designated as IN features make use of SCP capabilities. The features occupy about 10 pages of CRESS diagrams, and include billing, busy handling, call forwarding, call screening and conferencing. As a sample feature, figure 3 shows the template for Return Call. This is triggered when a phone starts ringing, storing the caller's number. It is also triggered by dialling. If the dialled number is 1471 (the UK code for Caller Return), it is replaced by the last caller.

The features are combined with POTS and the IN specification framework to yield about 2500 lines of SDL/PR. Depending on complexity, each feature has between 2 and 23 MSCs as use-case scenarios. The CRESS validation demonstrates well-known interactions among typical features. For example, Call Forwarding may interact with Call Screening: a caller may be forwarded to an undesirable number. Call Forwarding also interact with itself by causing a forwarding loop. However, CRESS is most valuable when analysing novel features for compatibility with existing ones.

4 Case Study: SIP Services

SIP (Session Initiation Protocol [10]) is an Internet standard for controlling sessions. In the context of Internet telephony, SIP is used to control voice calls. However SIP is a 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).

The goals of using CRESS with SIP include those for the IN. However SIP services are not yet so well understood, and SIP can support new kinds of services. Additional goals are therefore:

- to clarify what SIP services mean, and where they can be deployed
- to provide an architecture for defining SIP services
- to define a base of 'typical' SIP services.

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 calls should be redirected to reach a user). As a consequence, the CRESS model of services exposes all three elements. Unlike the IN, CRESS

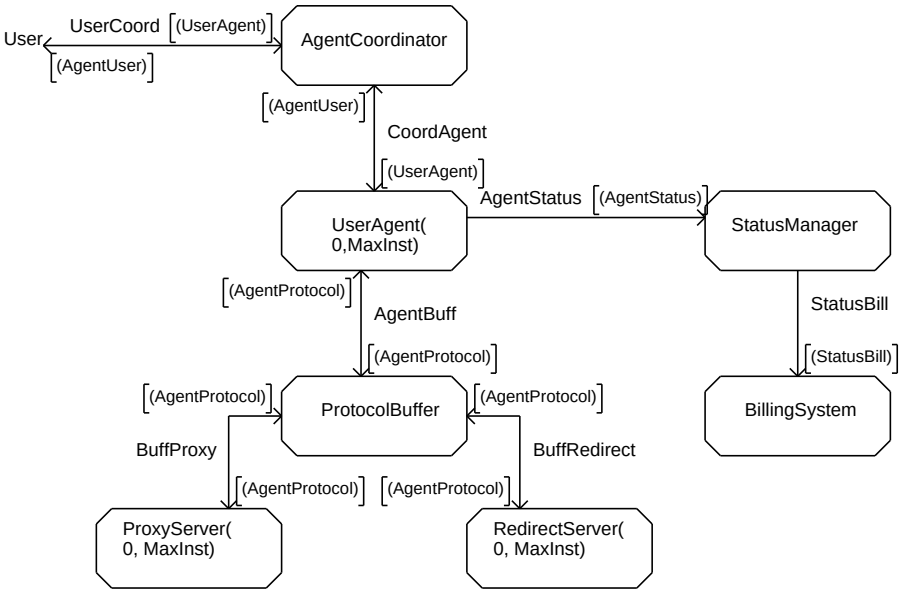

Fig. 4. CRESS SDL Structure for SIP

has to define half-call models for SIP (i.e. the behaviour of the caller or callee in isolation). For familiarity, service primitives follow telephony terminology. Thus a user is said to go *Off-hook* or *On-hook*, even though an actual phone may not be in use.

Ideally, SIP services would be described without internal protocol details. However SIP services are closely related to the protocol. CRESS is therefore obliged to include some aspects of this when describing features. This is not entirely desirable since it is then necessary to give the mapping between user actions and protocol actions. A reasonable compromise has been reached by mapping to an abstract view of SIP. This high-level interface is easily mapped onto the actual protocol.

The SIP specification framework resembles that for the IN. As it occupies 22 pages of SDL/GR, only the top level can be given here; more detail can be found in [12]. The main block structure is shown in figure 4. Internet telephony users communicate with the network via the *User* channel. The internal structure of the network comprises the following processes. In the SIP framework the *UserAgent*, *ProxyServer* and *RedirectServer* processes are all generated automatically; the other processes are fixed.

UserAgent consists of multiple instances up to the concurrent session limit *MaxInst*. The relevant CRESS diagrams define the root behaviour and features of a User Agent. It uses the abstract SIP interface to communicate with a *ProxyServer* or *RedirectServer* via the *ProtocolBuffer*.

ProxyServer consists of multiple instances up to the *MaxInst* limit. The relevant CRESS diagrams define the root behaviour and features of a Proxy Server.

RedirectServer consists of multiple instances up to the *MaxInst* limit. The relevant CRESS diagrams define the root behaviour and features of a Redirect Server.

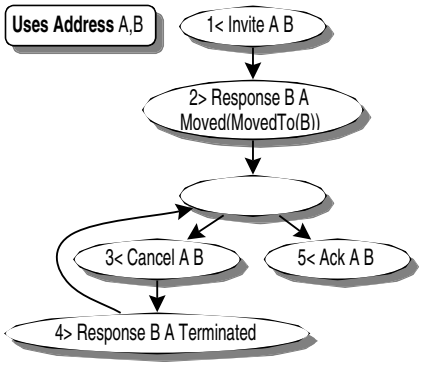

Fig. 5. CRESS Root Diagram for SIP Redirect Server

AgentCoordinator creates *UserAgent* instances and distributes user signals (much as the IN *CallCoordinator* does).

ProtocolBuffer distributes User Agent signals to Servers, and vice versa. It also creates instances of *ProxyServer* and *RedirectServer* as required. A Redirect Server is used if the called User Agent has call forwarding, otherwise a Proxy Server.

StatusManager maintains the status and profiles of all users (like its IN equivalent).

BillingSystem logs all billing information (like its IN equivalent).

Figure 5 is the CRESS root diagram for a Redirect Server. A sample feature appears in figure 6. This is triggered when a SIP response message *M* from callee *P* to caller *Q* is received by a Proxy Server. If the response is 'Busy Here', it is checked whether *P* has a forwarding address. If so an acknowledgement is sent from *Q* to *P*, and a fresh Invite is sent to the forwarding address of *P*. The new response message *M3* is also subject to a forward-on-busy check.

The kinds of SIP features described using CRESS are a subset of those for the IN. This is partly because SCP-based features are, of course, irrelevant for SIP: there are no centralised services. Because features may be deployed in three places (User Agent, Proxy Server, Redirect Server), there are variants of each feature. For example, call forwarding differs according to where it is deployed. Many IN features focus on handling busy conditions. In fact a SIP user may never be busy since there is no single telephone line/instrument that becomes occupied. For this reason, the notion of 'busy' is *defined* in CRESS. It may simply be as in the IN, i.e. only one call at a time is permitted. However it may be defined in a more complex manner to depend on the time of day, the caller, and the subject of the call.

As for the IN, root and feature diagrams for SIP are automatically combined and validated. A difference is that there are separate diagrams for User Agents, Proxy Servers and Redirect Servers. SIP features are characterised by MSCs and validated in the same way as for the IN. When standard IN features like call forwarding and call screening are reformulated for SIP, similar kinds of interactions are discovered. However, SIP lends itself to new kinds of features not found in the IN [9]. CRESS is useful to investigate new interactions arising from these.

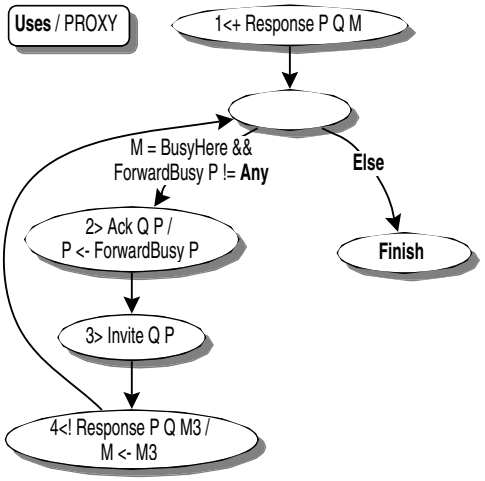

Fig. 6. CRESS Feature Diagram for Proxy Server Call Forwarding on Busy Line

5 Case Study: VoiceXML Services

VoiceXML (Voice Extended Markup Language [3]) draws on earlier scripting languages for interactive voice services. The underlying model of VoiceXML 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. Some actions may be governed by a condition or a count that specifies when the action is permitted. A script may throw an event, aborting current behaviour and activating a matching event handler.

The goals of CRESS for IVR (Interactive Voice Response) services differ somewhat from use with the IN or SIP:

- to explore the concepts of feature and feature interaction in an IVR setting
- to represent and analyse a range of generic and application-specific features
- to automate discovery of flaws in IVR applications.

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 databases 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. In the main, this means that a number of additional parameters may be given in a CRESS 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.

VoiceXML applications are often written as a number of documents containing a number of forms. However a VoiceXML application can be considered as a single document with a single form, and this is how it is ultimately 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. However, fields are deliberately not prominent in CRESS and are instead introduced implicitly.

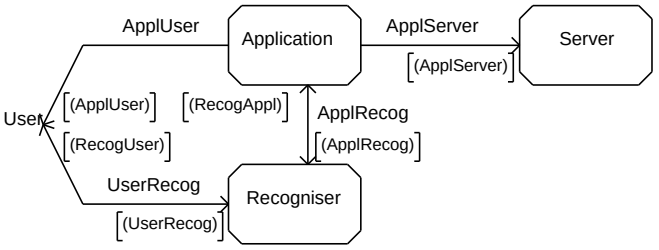

Fig. 7. CRESS SDL Structure for VoiceXML

The VoiceXML specification framework is much simpler than for the IN or SIP. The top-level structure is shown in figure 7. Telephone users communicate with the system via the *User* channel. Prompts and audio are 'spoken' to user, who 'speaks' in response. When using SDL, the goal is analysis and simulation so voice messages are represented by character strings. When using VoiceXML, voice messages indeed use audio.

Application is the main VoiceXML behaviour, created from the CRESS root and feature diagrams. The application may output ***Audio*** messages to the user. When a VoiceXML field needs to be filled in, the application sends information to the *Recogniser*: the required prompt and the grammar defining a valid response. The application may also submit information to the *Server*. The application may receive events from itself (***Throw***) or from the Recogniser. It therefore contains event dispatcher code that is generated statically from the service description.

Recogniser deals with the completion of fields. It issues a prompt and awaits a user response. This is checked according to the grammar, causing an application event like ***Filled*** (valid response), ***NoMatch*** (invalid response) or ***NoInput*** (input timeout).

Server represents a server supporting web and database access. Often the server just absorbs the results (e.g. writes them to a database). However, the server may return VoiceXML created on-the-fly. This cannot be handled except when VoiceXML is the target language. For SDL, CRESS handles the commonest cases of server scripts that produce no result (***Submit***), and scripts that compute some results (***Subdialogue***). Limited support is provided for the latter using a web adaptor written in C that links to the generated SDL.

IVR is quite different from the IN and SIP in having no unique root service. Rather, this depends entirely on the particular application. As a concrete example, figure 8 shows a quarry ordering application. It allows users to place telephone orders for a product (sand, gravel, cement) and the required weight. This information is then submitted for further processing to *order.jsp*, a Java servlet in the server. If the user requests help or says nothing, an explanation is given and the user is re-prompted.

VoiceXML was not defined with features in mind, though subdialogues act in a roughly similar manner. CRESS has been used to introduce a feature concept to VoiceXML. Generic CRESS features can be defined for use in a number of VoiceXML applications. Examples include defining VoiceXML behaviour (e.g. timeouts), asking for positive confirmation of an action (e.g. charging an account), and collecting customer

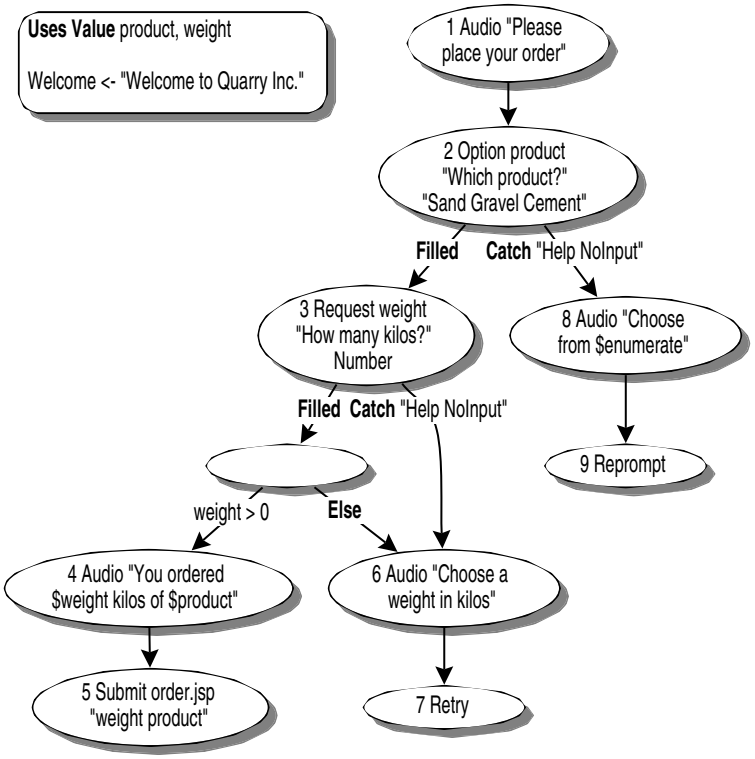

Fig. 8. CRESS Root Diagram for Quarry Ordering Application

identification (e.g. for an order). CRESS can also be used to define application-specific features. For example, a common 'welcome' feature has been defined for all quarry applications (ordering, checking delivery status, etc.).

Feature diagrams are combined with the root diagram for the basic application (e.g. for quarry ordering). VoiceXML is not as prone to feature interaction as the IN or SIP. Nonetheless, the CRESS approach allows interactions to be discovered in the same way. Perhaps more importantly, CRESS allows other kinds of service flaws to be identified through simulation. For example VoiceXML services may unintentionally loop, or there may be part of an application that it is not possible to execute. CRESS allows such problems to be discovered through reachability analysis of the generated SDL.

6 Supporting CRESS with SDL

This section gives an overview of how CRESS diagrams are translated into SDL. The translation strategy is designed for generality, even though a manual translation could be more straightforward in some cases. However the benefits of automated translation outweigh the occasionally indirect approach. CRESS is currently targeted at SDL 96 as this assures the widest tool support; small changes are required for SDL 2000.

6.1 CRESS Expressions

CRESS types are defined by the domain vocabulary. In the IN for example, they are ***Address*** (phone number), ***Boolean***, ***Message*** (service announcement) and ***Time***. ***Boolean*** and ***Time*** translate to the same types in SDL. ***Address*** translates to a digit string (including '#' and '*'), while ***Message*** translates to a set of literals like *EnterPIN*.

CRESS supports temporary variables, diagram variables and status variables. Temporary variables are declared for each type; for example *M0* to *M9* are available for message values. Variables may also be explicitly declared for use in a diagram, e.g. figure 8 declares *product* and *weight*. Temporary variables and diagram variables translate into ordinary SDL variables belonging to the corresponding process. Such variables are used directly in an expression, and are assigned in an SDL ***Task***.

Status variables are used to hold global information about a user, such as dynamic status or profile information. For example, SIP status includes whether the user is busy and the address of the user paying for the call. SIP profile information includes the user's forwarding number and the PIN used for charging a call to the user's account. Status variables are owned by a separate *StatusManager* process in the specification framework. They are ***Revealed*** by this process, and read elsewhere using ***View***.[1] Status variables are arrays of values with the same index and result types. For example, *StatusAB* is indexed by user address (*A*) to yield a boolean (*B*) result. Status is accessed by giving the variable name and indexes. For example, the CRESS expression *Busy A* is translated into the SDL expression ***View****(StatusAB)((. Busy,A .))*. Updating a status variable requires a signal to be sent to the status manager. This gives the variable name, indexes and new value. The CRESS assignment *Busy A* <- ***True***, for example, is translated into ***Output*** *UpdateAB(Busy,A,True)*.

Expressions are otherwise translated straightforwardly into SDL. Most CRESS operators have direct equivalents in SDL. A few (like *After*, which removes a prefix from a string) are defined in the specification framework. The generic value ***Any*** is translated to a special value of the corresponding SDL type (e.g. *AnyAddress*). ***Time*** in CRESS means the current time, and corresponds to ***Now*** in SDL.

6.2 Signals

A CRESS event node may contain parallel signals and also assignments. Although the CRESS translator has an option to deal with parallelism, for SDL this would have to explicitly unfold concurrency. (Sub-states in SDL 2000 might be an alternative.) The complex translation would hardly be worthwhile compared to the small increase in expressiveness. Concurrent inputs would also be very awkward to handle. Parallel signals as therefore translated as consecutive signals.

Assignments may be explicit in event nodes, or may be implied by the rules given in a rule box. For example in the IN, whenever phone *P* goes on-hook then the assignment *Busy P* <- ***False*** is implied (see the rule box in figure 2). Event assignments are handled like ordinary assignments.

[1] This is the only case that code generation differs for SDL 2000, which requires status variables to be read by a remote procedure call.

Output events are reasonably straightforward. For example in node 2 of figure 2, the CRESS event *DialTone A* is translated as the SDL signal ***Output*** *DialTone(A)*. A complication arises because CRESS diagrams may contain cycles. Since another node may loop back to the one containing an output, each output is preceded by a label constructed from the diagram name and node number. For node 2 in figure 2, the label is *POTS.2*. A loop back to this would then be translated as ***Join*** *POTS.2*. Sometimes CRESS has to generate two labels for the same SDL statement. This is unfortunately not allowed by SDL syntax, so the labels are separated by a dummy statement ***Task*** ␣.

Input events can be awkward to translate. In a simple case like node 5 of figure 2, the CRESS event *Answer B* is translated into the SDL ***Input*** *Answer(B)*. Since inputs must occur only at the start of a new state, this input is preceded with ***State*** *POTS.5*. If there are alternative inputs (e.g. nodes 7 and 10 of figure 2), one is used to label the new state but the transitions are individually labelled.

A CRESS diagram may also loop back to an input. The transition that follows this input is therefore labelled with *POTS.5*. A loop back to this must be translated as a repeated ***Input*** and then ***Join*** *POTS.5*. It might seem that this could simply be translated as ***NextState*** *POTS.5*. Unfortunately this does not work in general since loops are to individual inputs in a state. It is therefore necessary to branch to these inputs.

CRESS parameters like *A* and *B* are fixed when they are first input. However, SDL is prepared to input a completely different value. The CRESS translator therefore has an option to check if the parameter input is the same as the value expected. This is achieved by performing a data flow analysis of the CRESS diagram. If an input parameter is already fixed, an error occurs (***Stop***) if the actual parameter differs.

CRESS permits alternative branches to input the same signal. For example, nodes 7 and 10 of figure 2 both refer to *On-hook*. SDL does not permit alternative inputs to carry the same signal. The CRESS translator must therefore look at all alternative inputs and group those with the same signal. A single SDL ***Input*** then reads this signal. The signal parameter is checked, and the appropriate branch is taken (for *A* or *B* in this example).

The CRESS parser optimises diagrams before they are passed to a code generator. For example ***NoEvent*** nodes are removed where possible, and ***Else*** branches are moved to the end of the guard list. However it is not possible to remove a ***NoEvent*** node if it appears in loop (see figure 5). In such a case the ***NoEvent*** does not translate to any SDL, but the loop results in the same input being reached by different routes. A more complex state name must therefore be used. For example when node 5 is entered from node 4 in figure 5, the state label is *REDIRECT.5.REDIRECT.4*.

6.3 Actions and Guards

Application domains such as the IN and SIP require only inputs and outputs. However VoiceXML requires actions like ***Clear*** (clear form fields) or ***Throw*** (cause an event). These do not input or output and so are classed as actions. Actions are domain-specific, so their translation into SDL is also domain-specific. A ***Clear*** translates into a ***Task*** that sets field variables to undefined (initial) values. A ***Throw*** transfers control to the event dispatcher code. As for output, an action is preceded by a label in case some other part of a diagram loops back to it.

Expression guards are straightforward to translate. For example, *Free A B* in figure 2 translates to a ***Decision*** in SDL. As noted earlier, *Free* is macro-expanded to a check of *Busy*. Since this is a status variable, it is accessed using ***View***. An ***Else*** guard as in figure 2 corresponds to the *False* branch of a ***Decision*** in SDL.

An event guard is translated into an SDL ***Input*** of the corresponding signal. In VoiceXML, for example, this signal may come from the *Recogniser* process following the parsing of user input. An event guard may also be activated by a ***Throw*** within the application itself. A complication in VoiceXML is that event guards may be written at several levels: application, document, form and field. An event is handled at the closest enclosing level. In addition, events may have a hierarchical structure. Suppose that the quarry ordering application throws the *quarry.order.stock* event. If there is no event handler for this specific event, it may be caught by a handler for *quarry.order* or (failing that) *quarry* events. The CRESS translator manages this by maintaining the hierarchy of event handlers that apply at each level. (The hierarchy can be statically determined.) Events are interpreted by the event dispatcher according to the context, and may invoke platform, form or field handlers.

6.4 Sample Translation

Now that the basis of the SDL translation has been explained, it will be instructive to see what the generated code looks like. The following is the code created for figure 5. CRESS produces extensive comments that link the SDL back to the original diagrams. This is important because any flaw discovered in the SDL (e.g. an unreachable state) needs to be readily related to the original CRESS.

As mentioned earlier, state *REDIRECT.5.REDIRECT.4* is introduced since node 5 may be entered from node 4 via the empty node. As it happens, all the following inputs are available due to this loop. In this particular case, ***NextState*** *REDIRECT.5* could be the translation. But as explained previously, this is not always appropriate since only some inputs in a state may be repeated. The more indirect, but more general, solution is to handle the repeated ***Input*** signals and then ***Join*** the corresponding transitions.

```
Dcl A, A_0, B, B_0 Address;                                  /* call parameters */
Dcl A0, A1, A2, A3, A4, A5, A6, A7, A8, A9 Address;      /* temporary addresses */
Dcl M0, M1, M2, M3, M4, M5, M6, M7, M8, M9 Message;  /* temporary messages */
Dcl T0, T1, T2, T3, T4, T5, T6, T7, T8, T9 Time;          /* temporary times */

Start;                                                      /* start call instance */
   NextState REDIRECT.1;                                      /* for next input */

State REDIRECT.1;                                           /* ready for input */
   Input Invite(A,B);                                    /* REDIRECT input 1 */
    REDIRECT.1:
      Task ;                                          /* dummy label separator */
    REDIRECT.2:                                      /* REDIRECT output 2 */
      Output Response(B,A,Moved(View(StatusAA)((. MovedTo,B .))));
      NextState REDIRECT.5;                                    /* for next input */
```

```
State REDIRECT.5;                                   /* ready for input */
  Input Ack(A,B);                                   /* REDIRECT input 5 */
   REDIRECT.5:
    Stop;                                           /* end of behaviour */
  Input Cancel(A,B);                                /* REDIRECT input 3 */
   REDIRECT.3:
    Task ;                                          /* dummy label separator */
   REDIRECT.4:                                      /* REDIRECT output 4 */
    Output Response(B,A,Terminated);
    NextState REDIRECT.5.REDIRECT.4;                /* for next input */

State REDIRECT.5.REDIRECT.4;                        /* ready for input */
  Input Ack(A,B);                                   /* REDIRECT input 5 (again) */
    Join REDIRECT.5;
  Input Cancel(A,B);                                /* REDIRECT input 3 (again) */
    Join REDIRECT.3;
```

7 Conclusion

The CRESS notation has been briefly presented. CRESS offers the following benefits:

- CRESS applies to a variety of domains using plug-in vocabularies. The application of CRESS to IN, SIP and IVR have been briefly overviewed in this paper. 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 Service Description Language). It is advantageous to have a single notation that can be used in a number of domains.
- Since CRESS is graphical, it is more accessible to non-specialists. The diagrams are also more compact than their translations into various languages.
- CRESS is language-independent. So the same diagrams can be used for verification, validation and implementation. CRESS can be translated into formal languages for rigorous analysis, as well as into programming languages for realisation of services.
- CRESS explicitly supports features and feature composition. CRESS also provides domain-specific frameworks, e.g. for user profiles and billing. All these aspects would have to be re-specified if just a standard language were used.
- The CRESS tools are portable, and can be deployed on a variety of systems. CRESS can therefore be regarded as a platform-independent service creation toolset.

Although CRESS can be used in domains where features are not applicable, the most benefit is gained if this applies. CRESS should therefore be considered where a system can be thought of as having basic functionality plus additional features. CRESS has a simple notion of time as a monotonically increasing value. Any more realistic notion of time would require a target language with a concept of real time. CRESS therefore does not have explicit support for real-time systems.

CRESS itself is mainly a front-end for describing services. It is deliberately decoupled from analytic techniques. CRESS can therefore be used with any available technique, whether formal or informal. When used with SDL, reachability analysis and MSC validation are the most obvious techniques. But other techniques such as model-checking and theorem-proving are possible – limited only by the target language and its tool support.

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. 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.
3. V. Forum. *Voice eXtensible Markup Language*. VoiceXML Version 1.0. VoiceXML Forum, Mar. 2000.
4. N. D. Griffeth, R. B. Blumenthal, J.-C. Gregoire, and T. Ohta. Feature interaction detection contest. In K. Kimbler and W. Bouma, editors, *Proc. 5th. Feature Interactions in Telecommunications and Software Systems*, pages 327–359. IOS Press, Amsterdam, Netherlands, Sept. 1998.
5. 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.
6. ITU. *Intelligent Network – Q.120x Series Intelligent Network Recommendation Structure*. ITU-T Q.1200 Series. International Telecommunications Union, Geneva, Switzerland, 1993.
7. ITU. *Message Sequence Chart (MSC)*. ITU-T Z.120. International Telecommunications Union, Geneva, Switzerland, 2000.
8. ITU. *Specification and Description Language*. ITU-T Z.100. International Telecommunications Union, Geneva, Switzerland, 2000.
9. 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.
10. 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.
11. 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.
12. 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.