Modelling SIP Services Using CRESS

Kenneth J. Turner

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

Abstract. CRESS (CHISEL Representation Employing Systematic Specification) is a notation and set of tools for graphical specification and analysis of services. It is applicable wherever a system consists of base functionality to which may be added selected services. The CRESS notation is introduced for root diagrams, service diagrams, and rules governing their behaviour. It is shown how CRESS can represent services in SIP (Session Initiation Protocol). For analysis, service diagrams can be automatically translated into LOTOS (Language Of Temporal Ordering Specification) or SDL (Specification and Description Language). For scripting, translation is into CPL (Call Processing Language) or CGI (Common Gateway Interface). The structure of the portable CRESS toolset is explained.

1 Introduction

1.1 SIP

In telephony, a service means capabilities that are packaged and sold to end-users, while a feature is a self-contained aspect of a service. Most modelling approaches deal with features and how they can be composed to form larger services. A critical lesson from telephony is that services or features often interfere with each other in unexpected and undesirable ways – the so-called feature interaction problem [3].

IP Telephony or VoIP (Voice over IP) is a hot topic that has attracted significant commercial and research interest. The main standards deployed in this area are H.323 [6] and SIP (Session Initiation Protocol [9]). Although H.323 is more widely deployed and more mature, SIP is more flexible and better oriented towards providing new services. For example it is being used for presence and instant messaging services, and has been adopted for use in 3G mobile communications. However SIP services are a relatively new area, and service creation environments for SIP are only just emerging. Since SIP was designed from an Internet philosophy, it appears that some issues well known in telephony have not yet been transferred to the SIP domain. For example, feature interaction in SIP has received only limited attention [4].

This paper aims to clarify a number of important questions concerning SIP services:

- What is the nature of a SIP service? (see section 3.1)
- How might SIP services be modelled? (see section 3.2)
- How can SIP services be analysed and checked for compatibility? (see section 4.3)
- How can SIP services be prototyped? (see section 4.3)
- How can SIP models be used to create operational services? (see section 4.4)

D.A. Peled and M.Y. Vardi (Eds.): FORTE 2002, LNCS 2529, pp. 162–177, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Establishing a SIP session involves the User Agent of each end-user. Although User Agents can contact each other directly, it is preferable to establish sessions via Servers. The most flexible kind is a Proxy Server. This is often combined with a Registrar that receives notifications from users as to their current address (or even multiple addresses), thus supporting mobility. If a session invitation is sent to the user's Proxy Server, the Server can direct the request to the user's current location(s). Proxy Servers, and sometimes User Agents, also support service scripting. This allows the user to define call preferences, e.g. how they may be contacted by certain individuals, at certain times, or on certain subjects. A Redirect Server has the more limited role of returning a forwarding address to the initiating user, requiring a further session request to be issued.

Very briefly, SIP works as follows; consult the SIP standard [9] or a textbook [10] for more details. It may be helpful to note that SIP is patterned after HTTP. SIP commands ('methods') are confirmed by responses. SIP responses carry a numeric code and a text explanation. There are two broad classes of response: preliminary (e.g. session establishment in progress) and final (e.g. session setup succeeded or failed). A user initiates a SIP session by sending an *Invite*. The receiving user sends a *Response* that may accept or decline the session. The initiating user confirms receipt of this by sending an *Ack*. SIP negotiates the session media description, e.g. for audio and video. Even if the session is established via a Proxy Server, media data is sent directly between the users. To close a session, either party may send a *Bye* and confirm this with a *Response*.

1.2 CRESS

This paper discusses a development of CRESS (CHISEL Representation Employing Systematic Specification). CRESS is considerably evolved from its basis in CHISEL, which was developed by BellCore [1] for telephony services. CRESS is a notation and set of tools for graphical description and analysis of services [12]. It is graphical in order to improve its attractiveness to an industrial audience. CRESS has previously been used to model and analyse IN (Intelligent Network) services [12]. Adaptation of CRESS for SIP has been fairly straightforward, though different service models have been required.

Unlike CHISEL, CRESS allows modular service descriptions and permits much more flexible combination of services. CRESS also has 'plug-in' application domains, so it can be used outside traditional telephony. For example, the application to SIP is achieved by plugging in a different vocabulary and framework.

A formal language or a programming language could, of course, be used directly to model services. But CRESS aims to make service descriptions more accessible to non-specialists. CRESS diagrams are more compact than their translations into other languages, largely because they define services at a suitable level of abstraction. CRESS is operational in nature: it gives constructive, behavioural descriptions of services. Service diagrams derive their formal meaning through translation to a target language.

CRESS supports translation into implementation languages. In the context of SIP, service diagrams can be translated indirectly into C, or directly into CPL or CGI. CPL (Call Processing Language [8]) is used for SIP service scripting. SIP also has an HTTP-like CGI (Common Gateway Interface) that is also intended for SIP scripting.

CRESS is neutral with respect to the target application domain and target language. It can therefore be used as a front-end for some other (formal) approach. CRESS separates the

representation of services from their analysis, so it is open to various analytic techniques. The CRESS toolset is platform-independent, and so can be deployed widely.

Automated tool support has been developed to check the correctness of CRESS diagrams and to translate them into various (formal) languages. CRESS has a tightly defined notation that can be converted automatically into formal specifications. Formal analysis of services is based on a translation to LOTOS (Language Of Temporal Ordering Specification) or SDL (Specification and Description Language). This opens CRESS up to many formally-based techniques for analysing services and detecting interactions.

The CRESS approach confers a triple advantage. First, it is a comprehensible graphical notation for services. Second, it is automatically translated into a formal language for rigorous analysis. Third, it is automatically translated into implementation languages for deployment. Using CRESS with SIP partly aims to define and analyse services, but also to generate service scripts automatically. Although there are web-based tools for web scripting, these are close to the scripting language employed (typically CPL). CRESS aims to abstract from this, and to add rigour when defining services.

2 CRESS in General

2.1 Basic Diagrams

While reading this section, it may be helpful to refer forward to sample CRESS diagrams such as figures 2 and 6. A CRESS diagram is a directed, possibly cyclic, graph of nodes linked by arcs. A diagram describes the behaviour of a complete system, or of a service that is added to a base system. A basic node has a number and an associated event, e.g. *1 Off-hook A* to indicate subscriber *A* initiating a call. The node number is mainly for identification, but is used when services are combined. An event carries a signal like *Off-hook* and optional parameters like address *A*. If an event parameter has a known value it is used in the event, otherwise the parameter receives a value in the event.

Events are classified as inputs or outputs (as far as the system being specified is concerned). A composite node may contain several events in parallel, but these must be all inputs or all outputs. Input nodes normally alternate with output nodes along a path, but this is not a restriction. The signals in a node are used to determine if it is an input or output node. However, it is permissible to use the same signal for both input and output. (For example, a SIP User Agent may send or receive an *Ack*.) As a result, it may not be possible to determine the input/output kind of a node. In such a case, the kind of node is explicitly given by placing an input marker '<' or output marker '>' (mnemonic: from, to) after the node number. A system may merely relay signals (e.g. a SIP Proxy Server). To avoid many pairs of identical input/output nodes, the marker '&' (mnemonic: and) can be placed after the node number to mean consecutive input/output.

Each event may be associated with explicit assignments. These are normally separated by '/', but this symbol can be omitted (as in CHISEL) if there is no syntactic ambiguity. CRESS expressions allow the usual kinds of arithmetic, comparison, logical and set operators. If there is no ambiguity, parentheses are conventionally omitted around parameters, e.g. *Ack A B* means *Ack(A,B)*.

As an example of a composite event, a node might contain:

12> Stop Ring A B / Busy A <− **False** ||| Ack A B

This output node, numbered 12, occurs when *B* cancels a invitation to *A* because there is no answer. *A* stops ringing from *B*, and its status is recorded as no longer busy. In parallel, an acknowledgement of the cancellation is sent from *A* to *B*.

An 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. As in CHISEL, an empty node may explicitly contain **NoEvent**.

The arcs linking nodes may be plain arrows or may be labelled with a boolean condition as a guard. If branches of a choice are not guarded, the decision is determined by the events that follow. If branches are guarded, the decision is determined by the guard expressions. For convenience, an **Else** condition may be one of the alternatives.

A diagram must have a unique initial node. If cycles in a diagram mean that the initial node cannot be determined, an artificial **Start** node may be added to the diagram. A diagram may have several leaf nodes. Behaviour terminates here, or may cycle back to the initial node (at the specifier's discretion).

A large diagram may be split over several pages. Each section is lettered (to avoid confusion with the numeric node labels). An arrow symbol points to the next diagram section (e.g. *B*), which begins with this target label.

A CRESS root diagram describes the basic behaviour of a system. An important capability is being able to define additional service diagrams that modify the root diagram (or other service diagrams). Diagrams are combined syntactically using either of the methods discussed in section 2.3. Services are automatically combined with each other and with the root diagram to yield a composite system description. This approach is common in telephony, where the root diagram is POTS (Plain Old Telephone Service) and the service diagrams are additional services like call waiting or call screening.

2.2 Rule Boxes

A major informality in CHISEL concerns how variable values are changed by events. As seen in [5], such rules are written in English. In CRESS, a rule box gives a rigorous and machine-processible definition. Diagram variables are declared explicitly, e.g.:

Uses Address A **Address** B

An address is the identification of a user (e.g. a SIP address). Other variable types include *Boolean*, *Message* (voice message to a subscriber), *PIN* (Personal Identification Number), *Response* (response code) and *Time*. Temporary variables like address *A0..A9* and response *R0..R9* are implicitly available. **Any** stands for an indeterminate value (unknown or don't care).

In addition to diagram variables, CRESS supports status variables that capture user information and preferences. For example, a SIP invitation needs to know if the called party is busy or not. Status variables are typically indexed by address parameters. Thus *Busy P* indicates whether SIP address *P* is busy. Status variables are also used to hold user profile information such as what services have been set up, e.g. forwarding on busy.

Following the **Uses** statement, rules of various types can be given. For example, variable initialisation rules can be given. These are actioned only when the required values (*A* below) are known:

F := ForwardBusy A

Although the assignments triggered by an event can be written explicitly after the event, this clutters a diagram and becomes repetitious. Instead, CRESS allows rules to be formulated for assignments. For example when the calling party hangs up before the called party answers, the called party stops ringing and is no longer busy:

Stop Ring P Q / Busy P <– **False**

This is the same notation as used in an event node, except that the event parameters are place-holders. If an event matches the pattern above, *P* and *Q* are set to the actual parameters. An assignment rule may be overridden by an explicit assignment for the same variable in an event node.

Expression rewrite rules may be defined, e.g. 'idle' means 'not busy':

Idle P <– ∼Busy P

Any use of *Idle* is then transformed into a use of *Busy*. This kind of rule in fact defines a macro. Much more complex macros can be defined as shorthand notations (e.g. for a billing calculation). Macros can also be used to introduce named constants (e.g. for a time-of-day charge-band).

Occasionally useful, a signal transformation rule causes one signal to send another:

Start Billing P Q / LogBegin P Q P **Time**

meaning that when billing starts for a call from *P* to *Q*, the billing system gets a *LogBegin* signal. The call is from *P* to *Q*, with *P* paying, starting at the current time.

2.3 Service Diagrams

The CRESS notation introduced so far is essentially a convenient form of state transition diagram. Where CRESS makes a significant contribution is in its capabilities for combining services. The particular services deployed for each user are listed in a special configuration diagram (not illustrated here).

A service describes how it is inserted into another diagram. Typically this is the root diagram, although services may modify other services; for brevity, 'root diagram' in the following covers both cases. A service has a **Uses** statement to import the other diagrams it needs. If services depend on each other hierarchically, the subsidiary diagrams are imported automatically. In the simplest and commonest case, only the root diagram need be named: **Uses /** AGENT (i.e. the SIP User Agent diagram). Variables required by a service appear before '/' (though variables are often unnecessary). Service behaviour may be inserted into another diagram through splicing or instantiation.

Splicing Services. When a service is to be spliced it defines its attachment point in the root diagram, e.g. *AGENT 1*. This source node gives the diagram name and node number. (In fact, this is the main reason for having node numbers.) To attach to the first node of a diagram, the node 'number' is given as **Start**. The source node for a service may bind the values of service variables to those in the root diagram:

AGENT A<–X B<–Y 1

i.e. substitute variables *X* and *Y* in the service diagram for *A* and *B* in the root diagram when splice in behaviour starting at node *AGENT 1*.

Having located the point of attachment, a service defines what it alters in the root diagram. A node and its successors may be added to the root. Part of the root diagram may also be replaced in its entirety by identifying the original node, e.g. node *AGENT 1* and its contents *1 Off-hook A*. The effect is to replace this node and what originally followed it. Guards as well as event nodes may be added or replaced in a service diagram. A service may simply add behaviour that terminates in its own leaf nodes. More usually it continues with another part of the root diagram by referencing a target node like *AGENT 2*. A target node may also have variable bindings like a source node.

Service Templates. A service should be spliced if it applies just once to the root diagram. Another desirable condition for splicing is that the service has only very local effect on the root diagram. A number of the CHISEL services in [5] suffer from the problem of replacing large parts of the root diagram. For example CFBL (Call Forward on Busy Line) replaces about 80% of POTS, much of the diagram being similar to the original. CFBL can also apply more than once in a call. A call may be forwarded several times, for example, if successive forwarding numbers are busy. The original CHISEL diagrams can therefore really only be combined individually with the root diagram.

CRESS allows services to be defined more conveniently as templates. The initial template node states the event that may trigger it. For each matching trigger in the root diagram, an instance of the service is inserted. The template body has unique start and finish nodes, and may have its own leaf nodes. The start node is marked with '*' (mnemonic: any match) after the node number, while the finish node is an empty one (mnemonic: nothing further). The template is copied with substitution of actual parameters, and placed after the triggering node in the root diagram.

Sometimes it is not desirable to apply a template. For example anywhere there is an *Invite* to establish a SIP session, then forwarding services could apply. However some uses of *Invite* are not to establish a session, but rather to renegotiate the session media description. Template matching can be suppressed by placing '!' (mnemonic: don't match) after an event's node number.

2.4 Call Billing and Redirection

CRESS (CHISEL) is relatively unusual among modelling approaches in explicitly supporting billing. This is surprising since billing is a crucial aspect of services (for the operator at least!). In fact billing itself can lead to interactions. CHISEL has simple *LogBegin* and *LogEnd* events to denote the start and end of billing. The calling, called and paying parties are identified in these events. Normally the caller pays, but with freephone the callee pays. More complex arrangements can exist, e.g. the caller pays for part of the call and the callee pays the rest. Whereas billing is well understood in telephony or in the context of the IN (Intelligent Network), billing is still the subject of study for SIP. Nonetheless, CRESS already has the ability to deal with billing aspects.

Billing first checks which party will pay. Various services then have the opportunity to forward the invitation (unconditional, controlled by origin/time/subject, or depending

on destination busy). If the invitation is forwarded then the service chain is invoked again. This is necessary because the new destination may have different charging arrangements. By the time the invitation reaches its destination, it may have been forwarded several times. The billing for each redirection may also be different.

The *LogBegin*/*LogEnd* events of CHISEL are therefore insufficient. Although these are allowed by CRESS, *Start Billing*/*Stop Billing* events should be used instead. In fact, these are macro events that expand to *LogBegin*/*LogEnd* events for each redirection.

Establishing a SIP session is a major nexus for services to be invoked. Fortunately the service composition mechanism automatically handles the chaining of services. The designer can describe each service in isolation (i.e. in a modular manner), and their combination is automatic. In fact there are certain precedence rules that have to be enforced. For example billing must be considered before forwarding, and subscriber screening must be applied to the final SIP address obtained after forwarding. The CRESS tools automatically ensure that services are combined in a sensible order.

3 CRESS for SIP Services

3.1 SIP Model

CPL (Call Processing Language [8]) and SIP CGI (Common Gateway Interface) might appear to be adequate for defining SIP services. However they do not permit formal analysis in the way that CRESS does. The author also feels that CPL is too high-level (user-oriented, but at some distance from the protocol) while SIP CGI is too low-level (closely tied in with the protocol). The CRESS representation therefore aims at an intermediate level. The user interface is represented by familiar actions such as initiating, receiving and closing sessions. This is mapped to an abstract representation of the protocol interface such as issuing session invitations, re-establishing session parameters, and removing users from sessions.

Before applying CRESS to SIP, it is necessary first to decide how SIP should be modelled. Figure 1 shows the key SIP elements. The upper interface of each User Agent defines the service primitives that a SIP user sees. Like any service primitives, these are abstractions of an actual interface.

SIP primitives are deliberately named using conventional telephone terminology. For example a SIP user *A* starts or finishes a session with *Off-hook A*, though an actual telephone may not be used. A SIP session is requested from user *A* to user *B* by *Dial A B*, even though there may be no physical dial. User *A* is alerted to a potential session with user *B* by *Start Ring A B*, though in fact a pop-up window or some other indication may be used. *Announce A R* is used give user *A* a service response *R*, e.g. a progress signal like a SIP '180 Trying'. User *A* can refuse an invitation with response *R* by invoking *Reject A R*. If user *A* closes a session, user *B* is informed with *Disconnect B A*.

The lower interface of each User Agent deals with abstract SIP messages carried by the protocol. That is, each User Agent converts between service primitives and protocol messages. Many of the complexities of SIP are hidden. For example, most header fields are not represented, and issues such as retransmission or messages crossing are ignored. This is sufficient to represent major aspects of SIP for the purposes of service definition,

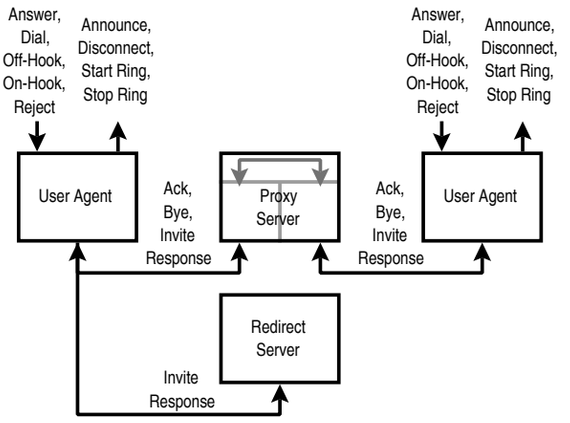

Fig. 1. Elements of SIP Model

but is certainly incomplete and not able to handle SIP services that depend on protocol details. Figure 1 does not show the complete repertoire of SIP methods (e.g. those for cancellation, registration, instant messaging, and third-party call control). A Proxy Server sits between users and handles SIP messages, acting like a gateway. A Redirect Server plays a much more limited role, sending a (forwarding) *Response* in answer to an *Invite*.

3.2 SIP Root Diagrams

Using CRESS it is possible to define a root diagram for the external view of a SIP session. As shown in figure 2, this describes the interface seen by the users in the session, omitting all protocol messages and actions by servers. The diagram is too detailed to explain here, but the brief summary of SIP in section 1.1 should help to make it comprehensible. This kind of model is appropriate for describing IN-like services such as call forwarding or conference calling. However, it is not so useful for SIP because SIP services can be deployed in a number of places: in User Agents and in Servers. In addition, SIP call services are also intimately bound up with the protocol. For these reasons, the CRESS treatment provides three root diagrams (User Agent, Proxy Server, Redirect Server) that show the mapping between user service primitives and protocol messages.

The User Agent model in figure 3 describes one end (half) of a session. The diagram is divided into originating (caller) and terminating (callee) parts. In fact, the SIP standard [9] does not provide a complete state machine description of the protocol. Figure 3 is therefore useful in its own right as an overview of SIP. But it *is* only an overview; it does not cover many details of the protocols (such as handling header fields or timeouts). The model is, however, sufficient to allow services to be added to the basic SIP behaviour. Many of the protocol messages can be sent or received by a User Agent or Server, and so are marked with '<' or '>' as discussed in section 2.1.

The Proxy Server model in figure 4 describes one session instance. Many of the protocol messages are relayed. The shorthand notation '&' mentioned in section 2.1 is therefore convenient. Initially a Proxy becomes involved when an *Invite* is received. It

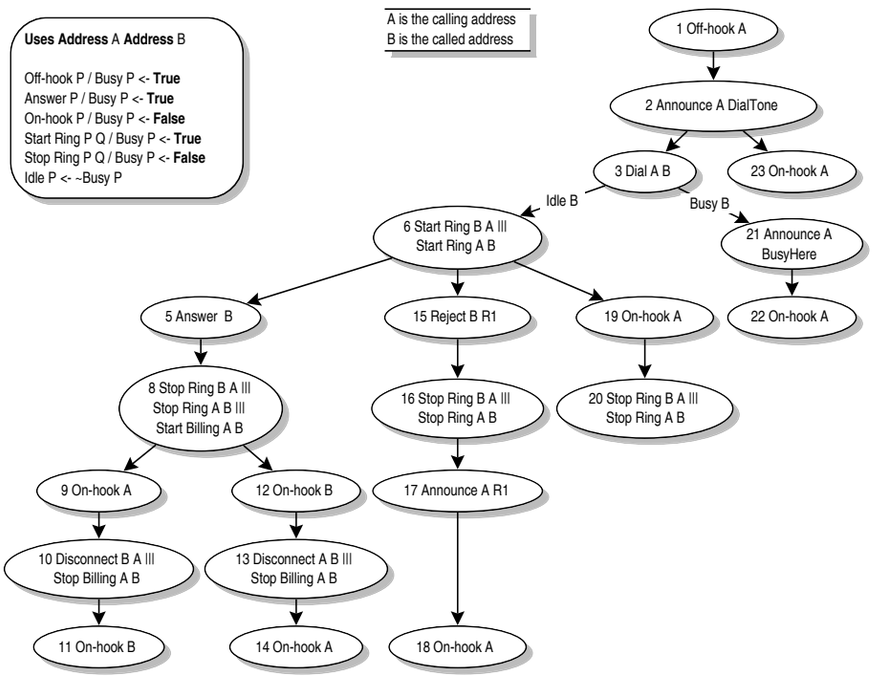

Fig. 2. CRESS Diagram for External View of SIP

passes this on, and relays any preliminary *Response* until there is a final response. If a session is established, the *Success* response is followed by the *Ack* to this. The Proxy now does nothing since the media streams are sent directly between the users. But it relays the closing *Bye* and the *Response* to this. If, however, session establishment does not succeed then the *Failed* predicate identifies a final but unsuccessful response. In this case, the Proxy waits for a *Response* and then continues from a new *Invite*.

The Redirect Server model in figure 5 is very straightforward. It repeatedly receives an *Invite*, and sends a *Response* with the *ForwardTo* address (if any) for the called user.

3.3 Sample SIP Services

A SIP service is considered to be a modification of the appropriate root diagram(s). An unfortunate characteristic of SIP services is that their definitions may differ according to where they are deployed. To give a flavour of the approach, the following shows some simple call control services. More complex services such as Conference Calling have also been represented in CRESS. CRESS is not limited to call control services. For example, it could be adapted for web-like services such as supported by SIP servlets.

Figure 6 shows how a subscriber screening service can be defined in a Proxy Server. The modifications are relative to the root diagram for the Proxy (figure 4). This service aims to screen out session requests from an undesired user. Screening is triggered when an *Invite* is received by the Server. If caller *P* is on the black-list (*ScreenIn*) for callee *Q*,

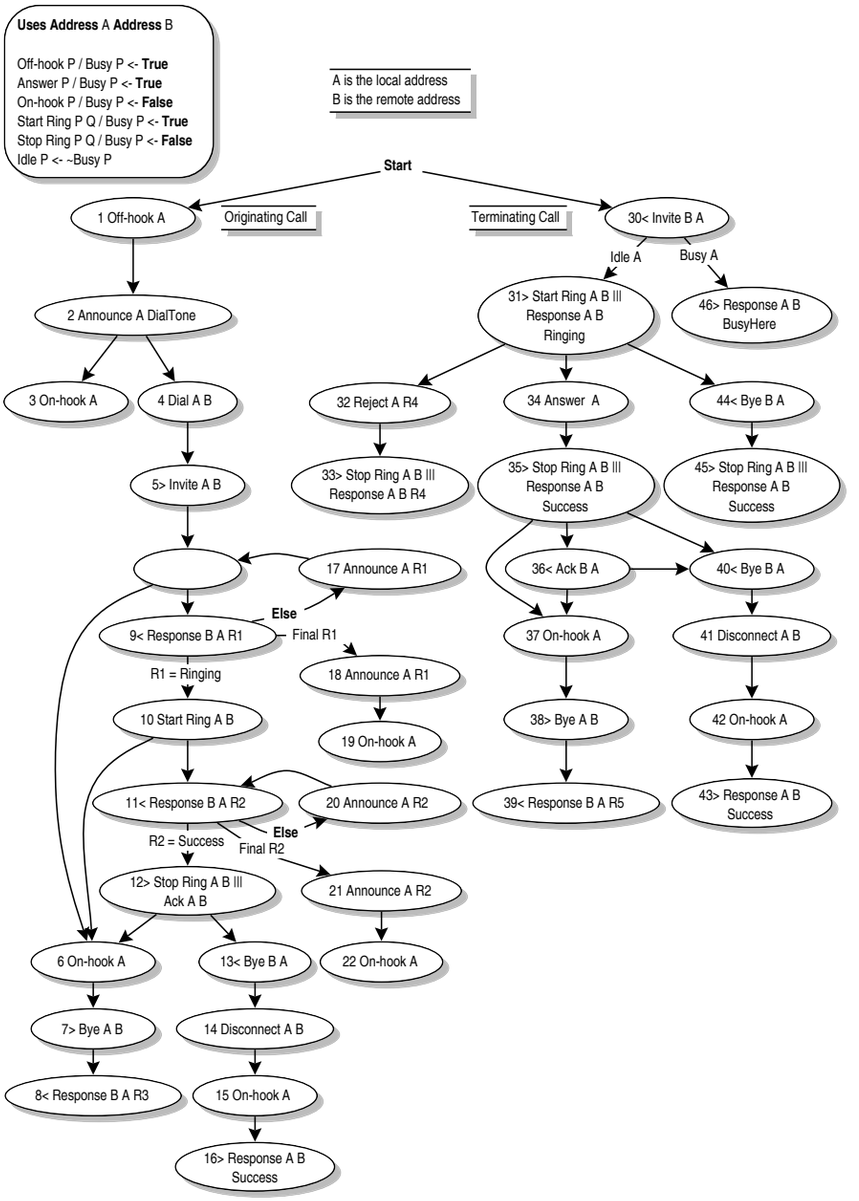

Fig. 3. CRESS Root Diagram for User Agent

the request is declined and session setup fails. Otherwise, the session request is processed as usual (following the template end-node, where Proxy behaviour continues).

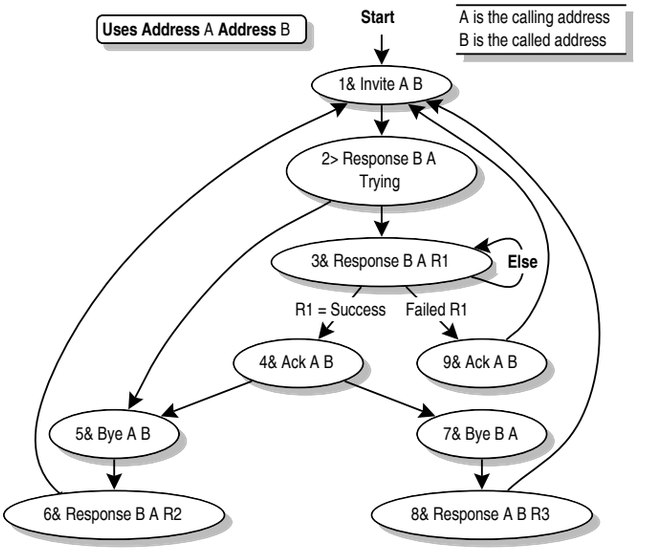

Fig. 4. CRESS Root Diagram for Proxy Server

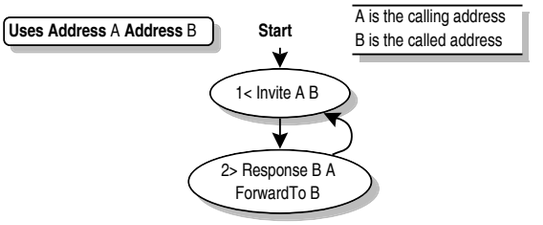

Fig. 5. CRESS Root Diagram for Redirect Server

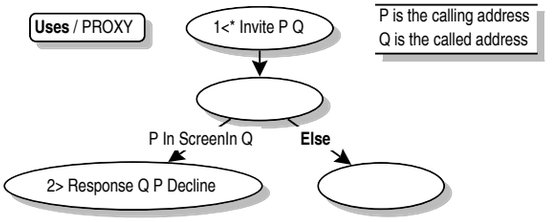

Fig. 6. CRESS Service Diagram for Proxy Server Incoming Call Screening

A second SIP service is shown in figure 7. This time the service is deployed in a User Agent, so the service modifies this root diagram (figure 3). If an *Invite* is received by a busy user *Q*, the User Agent will respond to caller *P* if there is a forwarding address

(*ForwardBusy*). The root diagram for a User Agent states what 'busy' means. According to the rule box of figure 3, busy simply indicates that the user is engaged in an existing session. However busy could be defined in other ways, e.g. if the user's diary shows an engagement for the current time. This could even depend on the caller and the subject. For example, a user in an existing SIP session might wish to accept a new *Invite* if it is from his manager, or if the subject is urgent.

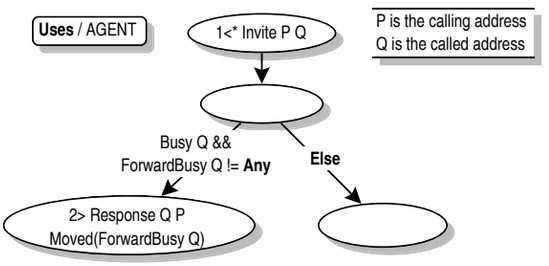

Fig. 7. CRESS Service Diagram for User Agent Forwarding on Busy

4 Tool Support

4.1 Toolset Structure

Figure 8 shows the CRESS tools. Symbols are shown doubled where there may be several files or several variants of a tool. The boxed area in figure 8 is the CRESS toolset. Outside this, the diagram editor and the target language tools are provided by others.

CRESS is designed for versatility and portability. It is therefore not bound to any particular diagram editor or target language. The tools are written in Perl 5, which runs on a wide variety of platforms. In total the toolset is about 5000 non-comment lines of code (five Perl scripts and five Perl modules). The code is quite intricate, and represents about 9 man-months of work. However the investment in the infrastructure has produced a general-purpose toolset of use in a variety of domains on a variety of platforms. To help others use and adapt the toolset, the code is extensively commented.

The author prepares CRESS diagrams using Lighthouse Design's *Diagram!* editor that runs on five different platforms. From preliminary investigations, it appears that a number of other diagram formats are suitable for CRESS (e.g. Adobe *Illustrator*, FrameMaker *MIF*, and *xfig*). Many diagram editors can produce output in well-defined formats. CRESS is thus not dependent on a particular diagram editor. In future, it is planned to develop a web-based editor for CRESS diagrams.

CRESS is also not bound to any particular target language. For formal analysis, translation to LOTOS and to SDL is supported. E-LOTOS was studied as a target language as it confers some advantages relative to LOTOS. However E-LOTOS tools are only at an early stage, so E-LOTOS is not yet a target for CRESS. For service scripting, CRESS diagrams can be translated to CPL or CGI. CPL is an intentionally restricted language, but SIP

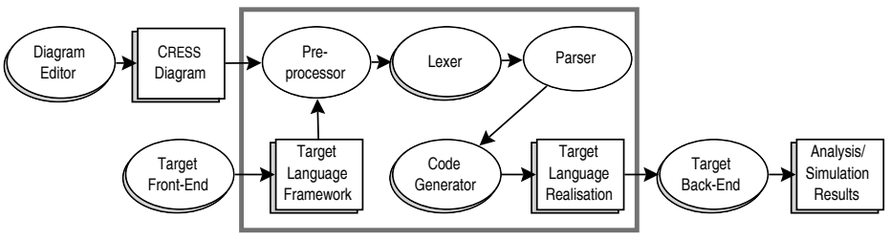

Fig. 8. CRESS Toolset

CGI scripts can do almost anything. CRESS translates diagrams into a stylised form of Perl for use as CGI scripts.

The target language framework is created using the target development environment. Since the framework is fixed for a given domain and target language, it can be provided as standard. IN and SIP frameworks for LOTOS and SDL are currently available. SIP frameworks are available for CPL and CGI. The framework provides the architecture in which the services are embedded. For example, the SIP framework defines the behaviour of the status manager and the billing system.

4.2 Toolset Usage

The designer prepares CRESS diagrams using a convenient editor. The designer is assumed to have a suitable development environment for the target language. Most development environments allow pre-processing. A simple command or button click can activate the CRESS toolset automatically. The CRESS pre-processor scans the target language framework for CRESS macro calls:

Cress(Types)	(* generate domain-dependent types *)
Cress(Profiles)	(* generate user profile information *)
Cress(Services)	(* combine root diagram and services *)

Each of these is expanded to the corresponding definitions in the target language. The types (and associated functions) are partly fixed and partly dependent on the domain of application. Since (status) variables and signals are defined in tables loaded into the tools, a change of domain is easy to arrange. The variable/signal tables are used while checking diagrams, and are also used to generate the domain-specific types.

Each CRESS macro call is expanded using the toolset. The lexer appropriate to the diagram editor is called to build a rule list and event node graph for each diagram. The parser is common, and checks the syntax and static semantics of each diagram separately. The parser then combines the root and service diagrams, performing further consistency checks. A number of optimisations are carried out on the graph to make code generation more efficient. For example empty (**NoEvent**) nodes are removed where possible, **Else** is moved to the end of alternative guards, and alternative inputs are ordered by signal name. Finally the parser hands the graph to the appropriate code generator that outputs in the target language. To the target development environment, a single (albeit very complex) step of pre-processing has taken place. The translators have an option to produce detailed comments in the generated code, thus simplifying maintenance of the tool output.

Once the target language specification has been generated, the language back-end tools can be used to simulate, analyse or implement the specification. Both LOTOS and SDL can be used for single-step or automated simulation. They can also be used for state space exploration. LOTOS has advanced analysis tools for state space minimisation, equivalence checking, model checking, etc. Both LOTOS and SDL can be compiled to usable C for implementation. If a CPL or CGI script is generated from a CRESS diagram, the script is directly usable by a SIP User Agent or Proxy Server.

4.3 Formal Analysis

The emphasis in CRESS so far has been on the representation and formalisation of services. CRESS work on detecting service interactions has been limited to date. The following presents preliminary work, but in fact techniques developed by others can be adapted for use with the specifications generated by CRESS.

To check the correctness of services, each was simulated on its own when combined with the appropriate root diagram. Human judgment was used in deciding significant execution paths, the aim being to execute each path at least once. This procedure built confidence in the service descriptions as well as creating a set of validation scenarios.

For simple services the number of paths to check is small. For complex services, the number of interesting paths is just manageable. Conference Calling, for example, has 23 significant paths.

Validation scenarios are generated to characterise the expected behaviour of services in isolation. The scenarios can then be encoded using the ANTEST (ANISE Test) language developed by the author for ANISE [11]. Briefly, ANTEST is a flexible validation language that expresses tests in terms of user-visible behaviour. Acceptance tests (behaviour must happen) and rejection tests (behaviour must not happen) can be written. Tests may have sequential or concurrent behaviour. Alternatives are permitted, and behaviour can be made conditional on a service being present for a SIP address. In fact, ANTEST is used to encode comprehensive use-case scenarios that synthesise the individual executions obtained through simulation.

The ANTEST tool automatically translates validation scenarios into the target language (currently only LOTOS), and automatically runs them in parallel with a specification. When services are combined individually with a root diagram, this merely confirms the validity of the scenarios. More importantly, the validation scenarios can be run with *all* services deployed. A common interpretation of service interaction is that a service fails to perform as expected when other services are present. The manifestation of service interaction when using ANTEST is either deadlock or non-determinism. Deadlock means that one service prevented another from working. Non-determinism means that an ambiguity arose.

In future work, the author intends to apply techniques developed from protocol conformance testing [7] to derive use-case scenarios automatically. Another avenue to be explored is the use of model-checking to verify that service properties are preserved in the presence of other services. The current validation approach can check only specific scenarios. Model-checking should allow such properties to be proved in general. An approach based on symbolic model-checking [2] looks an attractive possibility, though tools are currently under development.

4.4 Service Scripting

CRESS diagrams can be translated into CPL scripts. For example, User Agent forwarding on busy defined in figure 7 corresponds to the following CPL:

```
<incoming>                                              # incoming call
  <proxy>                                                 # forward call
    <busy>                                              # if callee is busy
      <location url = "sip:forward@domain">      # to forward@domain
    </busy>
  </proxy>
</incoming>
```

It should, however, be noted that it is not possible to translate an arbitrary CRESS diagram into CPL since the latter is much more restrictive. Instead, only certain patterns of CRESS diagram can be turned into CPL scripts. Furthermore, a CRESS diagram is generic and must be instantiated with user configuration data during translation to CPL.

CRESS diagrams can also be translated into CGI scripts using stylised Perl. For example, User Agent screening defined in figure 7 corresponds to the following CGI/Perl:

```
if ($Method == $Invite) {                               # if method is Invite
  if ($Busy ($Q) &&                                     # if callee is busy and
      $ForwardBusy ($Q) != $Any) {                  # has a forwarding address
    &Response ($Q, $P,                                    # respond to caller
      &Moved ($ForwardBusy ($Q)))                   # with forwarding address
  }
}
```

It is much easier to translate a CRESS diagram into CGI/Perl since this is much more expressive than CPL. As the above example shows, the translation relies on a Perl framework (not shown) that establishes variables (Perl prefix '$') containing the SIP method and its parameters. These are obtained from the CGI script environment variables. In addition, a number of Perl subroutines (Perl prefix '&') are pre-defined to handle call processing. For example, *Response* returns the CGI script output to the Proxy Server.

5 Conclusion

It has been seen that CRESS is a graphical language for specifying systems with a base functionality and additional services. The elements of the notation have been introduced for root diagrams, service diagrams and rules. The particular contribution of CRESS is its ability to describe and combine services in a flexible and automatic manner. A portable toolset enables thorough checking and translation of diagrams to various languages for formal analysis (LOTOS and SDL) and for scripting (CPL and CGI/Perl).

Preliminary work has been presented on service interaction detection using the specifications generated by CRESS. Future developments will include web support for graphical service description and analysis. More complete interaction detection techniques will also be developed. Although CRESS has been illustrated with SIP services, it is applicable to a number of other problem domains such as the IN.

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. M. Calder and C. E. Shankland. A symbolic semantics and bisimulation for full LOTOS. In M. Kim, B. Chin, S. Kang, and D. Lee, editors, *Proc. Formal Techniques for Networked and Distributed Systems (FORTE XIV)*, pages 184–200. Kluwer Academic Publishers, London, UK, Sept. 2001.
3. E. J. Cameron, N. D. Griffeth, Y.-J. Lin, M. E. Nilson, W. K. Schnure, and H. Velthuijsen. A feature-interaction benchmark for IN and beyond. *IEEE Communications Magazine*, pages 64–69, Mar. 1993.
4. B. El Ouahidi and M. Bouhdadi. Internet/telecommunications integration: Towards IN-capable SIP networks. *Networks and Distributed Systems (Réseaux et Systèmes Répartis)*, 12(2):259–280, Oct. 2000.
5. 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.
6. ITU. *Packet-Based Multimedia Communication Systems*. ITU-T H.323. International Telecommunications Union, Geneva, Switzerland, 2000.
7. Ji He and K. J. Turner. Protocol-inspired hardware testing. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, *Proc. Testing Communicating Systems XII*, pages 131–147, London, UK, Sept. 1999. Kluwer Academic Publishers.
8. J. Lennox and H. Schulzrinne, editors. *CPL: A Language for User Control of Internet Telephony Services*. Internet Draft CPL-01. The Internet Society, New York, USA, Mar. 2000.
9. J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnson, J. Peterson, R. Sparks, M. Handley, and E. Schooler, editors. *SIP: Session Initiation Protocol*. RFC 2543 *bis* 09. The Internet Society, New York, USA, Feb. 2002.
10. H. Sinnreich and A. B. Johnston. *Internet Communications using SIP*. John Wiley and Sons, Chichester, UK, 2001.
11. 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.
12. 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.