Kenneth J. Turner. Modular Feature Specification.
In Tom Gray, editor, Proc. MICON 2001,
Mitel Networks Corporation, Kanata, Canada, August 2001

Modular Feature Specification

Kenneth J. Turner
Computing Science and Mathematics, University of Stirling
Stirling FK9 4LA, Scotland
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 features. It is applicable wherever a system consists of base functionality to which are added optionally selected features. The CRESS notation is introduced for basic diagrams, feature diagrams, and rules governing their behaviour. Although telephony is used to illustrate the approach, CRESS is not limited to this domain. The structure and use of the portable CRESS toolset is explained. CRESS can generate code for a variety of target languages. The strategy for translation to LOTOS is presented, along with some techniques for analysing the generated specifications.

Keywords: feature, IN (Intelligent Network), LOTOS (Language Of Temporal Ordering Specification), SDL (Specification and Description Language), service, telephony

1 Introduction

1.1 Overview of CRESS

CRESS (CHISEL Representation Employing Systematic Specification) is a notation and set of tools for graphical specification and analysis of features. CRESS is designed as a flexible notation for describing and combining features. It is graphical in order to improve its attractiveness to an industrial audience. Automated tool support has been developed to check the correctness of diagrams and to translate them into various (formal) languages. The toolset is portable and can run on a variety of platforms with a variety of front-end diagram editors and back-end target languages.

CRESS is based on the CHISEL notation developed by BellCore [1] for describing telephony features. An important reason for choosing CHISEL as the basis is that it was developed to meet industrial needs. As far as practicable, CRESS is a strict (if substantial) extension of CHISEL. This means that practically any CHISEL diagram can be used with CRESS, although it is highly desirable to use the new capabilities of CRESS. CHISEL has been significantly extended and generalised in CRESS. More importantly, CRESS has a tightly defined notation that is translated automatically into formal specifications – currently LOTOS (Language Of Temporal Ordering Specification) and SDL (Specification and Description Language).

A CRESS root diagram describes the basic capabilities of a system. The notation resembles a state transition diagram. The main capability of CRESS is describing additional feature diagrams that modify the root diagram (or other feature diagrams). Features are automatically combined with each other and the root diagram to yield a composite system description. This approach is common in telephony, where the root diagram corresponds to POTS (Plain Old Telephone Service) and the feature diagrams correspond to additional services like CW (Call Waiting) and TWC (Three-Way Calling).

CRESS is not fixed in the telephony domain, although this is the main application to be presented in the paper. It can be used to describe any system that consists of base functionality and additions to this. This situation is quite common; examples arise in defining car accessories, electronic mail [6], lift control [11], object-oriented development, operating systems and word processors. Combining a number of features often leads to the feature interaction problem [3], whereby features that are conceived in isolation do not interwork properly. One of the motivations for translating CRESS into a formal specification is that it permits rigorous analysis of feature interaction problems.

The work has achieved a graphical and flexible representation of modular features. It is applicable to a variety of domains, although it has been validated in the field of telephony. The approach is neutral with respect to the target language, and so can be used as a front-end for any formally-based approach. The toolset is portable and can be used on a range of platforms.

1.2 Previous Work

The initial version of CRESS was described in [14]. The emphasis in that paper was on formalising CHISEL diagrams. The main limitation was that feature descriptions were not very modular. Features could be described and analysed individually, but their automated combination was severely limited. The work reported in the present paper has addressed the modularity problem. Feature descriptions are now modular and more compact. Feature combination has also been made much more flexible. Interestingly the improvements have require relatively small changes to the graphical notation, though the underlying tool support has had to be substantially extended.

The notion of feature composition has been around for some time. Architectures such as [8, 10, 15] have been devised to allow features to be treated as building blocks. In some approaches, the architectural descriptions are translated to formal specifications that permit further analysis. For example, [2] describes use-case maps for telephony features that are hand-translated into LOTOS. [12, 13] and other papers illustrate the author's work on ANISE (Architectural Notions in Service Engineering). ANISE builds features and services in a hierarchical fashion, and then translates them to LOTOS for simulation and interaction analysis.

CRESS intentionally separates the representation of features from their analysis. Those who apply formal methods to feature interaction often have to waste time re-discovering and re-specifying features in their formalism. CRESS is intended as a common front-end for other languages. CRESS also does not need to develop its own approach to feature interaction since this is handled by other tools that take a specification as the starting point. As an example, the University of Ottawa [4] have developed analysis techniques that can be used with the output of CRESS. In fact the Ottawa team wrote their specifications by hand, but as reported in [14] the LOTOS generated CRESS is more compact and more readable. And of course, automatically generated specifications are much easier to maintain through changes to the feature diagrams.

2 CRESS Notation

This section provides a brief overview of the CRESS notation. CRESS is illustrated with reference to telephony and the IN (Intelligent Network); specifically, features taken from [5] are used as examples. CRESS is not, however, limited to telephony. The tools are table-driven so that CRESS may be used in a variety of other domains.

2.1 Basic Diagrams

A CRESS diagram is a directed, possibly cyclic, graph of nodes links by arcs. A basic node has a number and an associated event, e.g. *1 Off-hook A* to indicate subscriber *A* picking up the phone. The node number is only for identification, but plays a role when features are introduced. An event carries a signal such as *Off-hook* and optional parameters like *A*. If an event parameter has a known value it is supplied in the event, otherwise it receives a value as a result of 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. 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. An example of a composite event is:

14 Stop Ringing B A / Busy B $\leftarrow$ False ||| Stop AudibleRinging A B

An empty node, meaning no event occurs, is occasionally useful as a connector. It can be used to join a number of preceding and following nodes as a more compact way of linking all the nodes directly. As in CHISEL, an empty node may be explicitly labelled as **NoEvent**.

The arcs linking nodes may be plain or may carry a boolean condition as a guard. If the branches of a choice are not guarded, the decision is determined by the events that follow. If the branches are guarded, the decision is determined by the guard expressions. For convenience, an **Else** condition may be used as 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 either terminates here or cycles back to the initial node (as a design choice).

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.

2.2 Rule Boxes

A major informality in CHISEL concerns how variable values are changed by events. As may be seen from [5], such rules are generally expressed in English. In CRESS, a rule box provides a formal definition. The variables used by a diagram are defined explicitly, e.g.:

Uses Address A Address B

An address is the identification of a user (e.g. a phone number). Other variables types include *Boolean*, *Message* (voice message to a subscriber), *PIN* (Personal Identification Number) and *Time*. Temporary variables like address *A0..A9* and message *M0..M9* are implicitly available. **Any** stands for an indeterminate address – an unknown or don't care value.

In addition to diagram variables, CRESS supports status variables that capture globally significant information. For example, a phone call needs to know if the called party is busy or not. Status variables are typically indexed by address parameters. Thus *Busy P* indicates whether phone number *P* is busy. Status variables are also used to hold user profile information such as what features have been subscribed to, e.g. *CallWaiting P*. Following the **Uses** statement, rules of various types can be given.

Variable initialisation rules can be given, e.g.:

F := ForwardBusy A

CRESS delays such initialisations until the required values (*A*) are known.

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:[1]

Stop Ringing P Q / Busy P $\leftarrow$ False

This is the same notation as used in an event node, except that the event parameters are placeholders. If an event matches the pattern above, *P* and *Q* will be 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. A simple example is that a line being idle means it is not busy:

Idle P $\leftarrow$ $\sim$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. Macros can also be used to introduce named constants.

Finally, it is occasionally useful to define a signal transformation rule that causes one signal to generate another:

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

With this brief introduction to CRESS, the description of POTS in figure 1 should be comprehensible. The rule box is the rounded rectangle at the top left of the figure. The text between horizontal bars at top centre is a comment. The event nodes are shown as shadowed ovals. In fact the shapes of the symbols in a diagram are irrelevant as long as they are distinct and consistent. Other forms of comment are possible such as audio, graphical and textual attachments.

[1] CRESS needs only simple rules for *Busy* because its boolean variables are 'sticky'. For example if a variable is assigned *True* twice, it needs to be assigned *False* twice before it reverts to *False*.

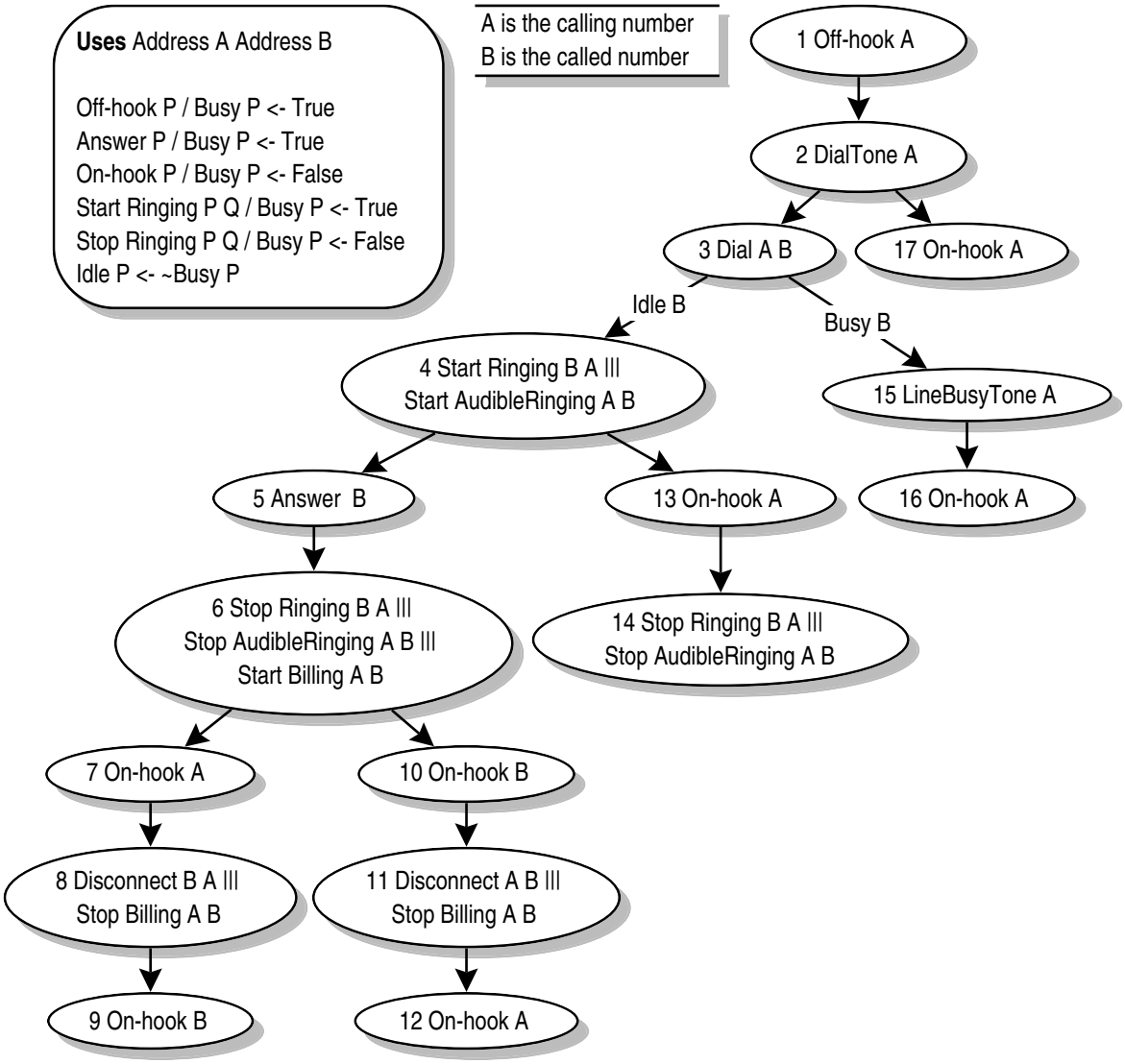

Figure 1: CRESS Root Diagram for the Plain Old Telephone Service

2.3 Feature 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 defining and combining features. A feature describes how it is inserted into another diagram. Typically this is the root diagram, although features may modify features; for brevity, 'root diagram' in the following covers both cases. A feature has a **Uses** statement that imports the other diagrams it needs. If features 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** / POTS. Any variables required by a feature are declared before the '/' (though this part of a **Uses** statement is often empty). Feature behaviour may be inserted into another diagram through splicing or instantiation.

2.3.1 Splicing Features

When a feature is to be spliced it defines its attachment point in the root diagram, e.g. *POTS 7*. 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 feature may bind the values of root diagram variables to those in the feature:

POTS A<–X B<–Z 7

Having located the point of attachment, a feature defines what it alters in the root diagram. A node and its successors may added to the root. Part of the root diagram may also be replaced in its entirety by identifying the original node, e.g. node *POTS 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 feature diagram. A feature may simply add behaviour that terminates in its own leaf nodes. More typically it continues with another part of the root diagram by referencing a target node like *POTS 2*. A target node may also have root/feature variable bindings like a source node.

Figure 2 shows how the feature INTL (IN Teen Line) is spliced into POTS. The idea of INTL is that use of the phone between certain hours requires a PIN to be entered (e.g. to prevent teenagers from calling within peak hours). This feature makes use of signals between the telephone switch and the SCP (Service Control Point, as used in the IN). SCP signals are somewhat complex and irregular in their structure, though they always start with the triggering phone number. This may be followed by the calling number and the called number. Other parameters may include the current time, the paying party or a forwarding number. The exact details of SCP signals are given in [5].

The INTL feature is defined to replace POTS node 1 and its transition to node 2: the initial off-hook progressing to dial tone, as shown in figure 1. When the phone goes off-hook, the SCP is alerted: INTL events 1 and 2. If INTL is not enabled for this phone or for the current time, the SCP allows the call to continue: INTL event 13, POTS event 2. Otherwise a voice message *M1* (e.g. 'Enter PIN') is provided by the SCP and announced to the user: INTL events 3 and 4. The user may hang up and abort the call attempt: INTL events 5 and 6. Alternatively, the user may dial a PIN in the form of a phone number *A1*: INTL event 7; see section 2.3.2 regarding the use of '7!'. The PIN is sent as a resource value to the SCP. If the PIN is correct, the SCP allows the call to continue: INTL event 13, POTS event 2. Otherwise the SCP causes a voice message *M2* (e.g. 'Wrong PIN') to be announced, and the call is forcibly terminated: INTL events 9 to 12.

2.3.2 Feature Templates

A feature should be spliced if it applies just once to the root diagram. Another necessary condition for splicing is that the feature should have only very local effect on the root diagram. A number of the CHISEL features in [5] suffer from the problem of replacing large parts of POTS. For example CFBL (Call Forward on Busy Line) replaces about 80% of POTS, much of diagram being similar to the original. Worse still, CFBL could apply several times in a call. A call may be forwarded several times, for example, if successive forwarding numbers are busy. TWC (Three-Way Calling) sets up two call legs, and CFBL may apply in each case. The original CHISEL diagrams can therefore really only be combined individually with POTS. The

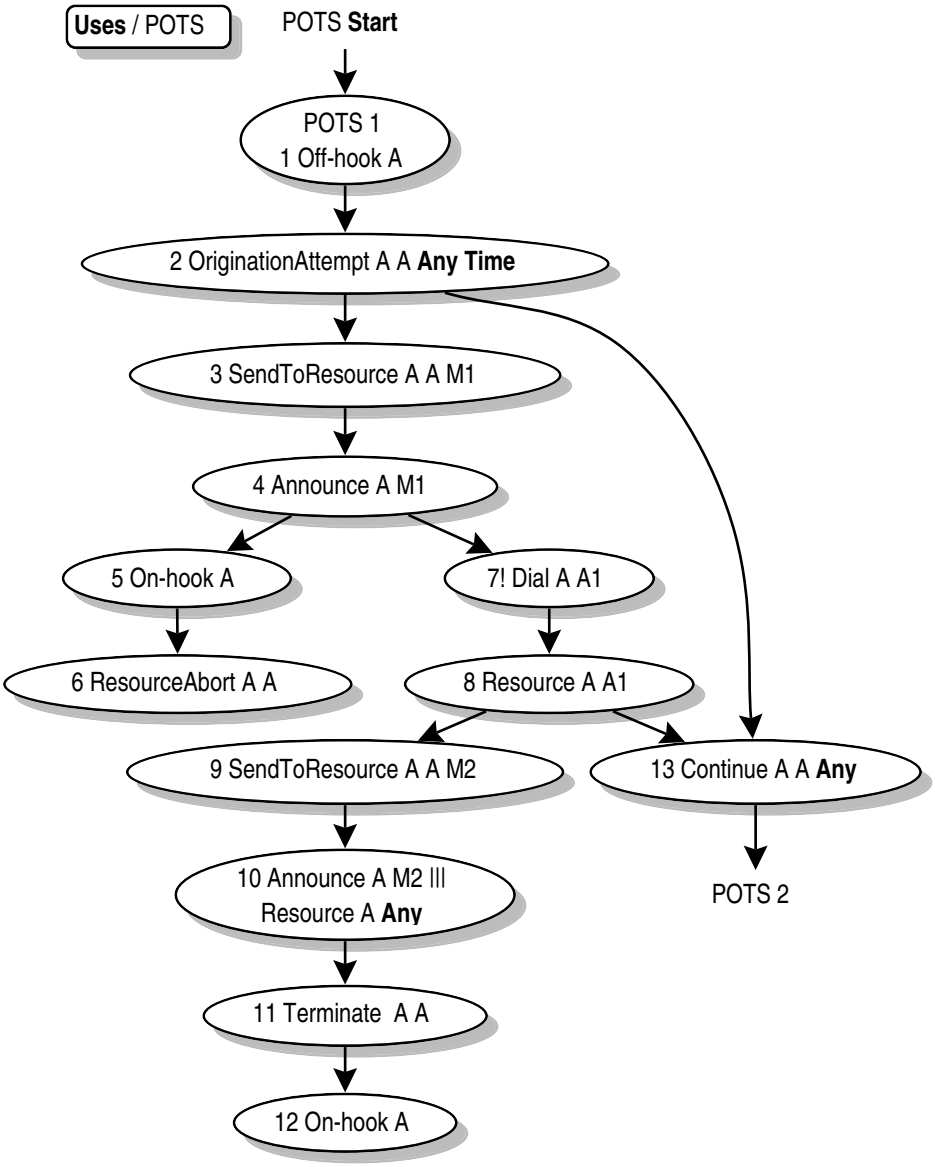

Figure 2: CRESS Feature Diagram for IN Teen Line

diagrams are modular in the limited sense of being self-contained, but are not modular in the sense of being parameterised and re-usable.

CRESS therefore permits features to be defined as templates. A feature template defines the pattern of its behaviour. The initial feature node defines the event that may trigger it. For each matching trigger in the root diagram, an instance of the feature is inserted. The template body requires unique start and finish nodes; if necessary, empty nodes can be used for this purpose. The template is copied with substitution of actual parameters and placed after the triggering node in the root diagram.

Figure 3 shows CND (Calling Number Delivery) as a feature template. This allows the destination to see the number of the caller. The asterisk in the triggering node (‘*1**’) indicates a wild card event: CND event 1 will match any node in the root diagram that contains a *Start Ringing* event with two parameters. *P* and *Q* are formal parameters of the template,

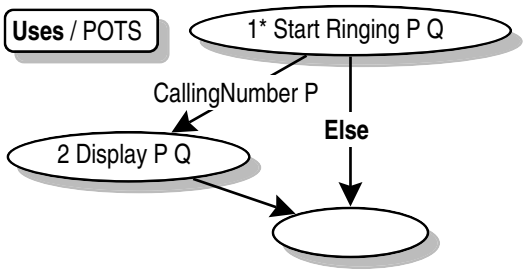

Figure 3: CRESS Feature Diagram for Calling Number Delivery

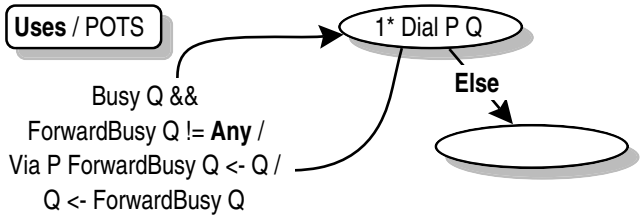

Figure 4: CRESS Feature Diagram for Call Forward on Busy Line

matched to the actual parameters in the triggering event. If the number being rung has caller display (*CallingNumber P*), the number of the caller (*Q*) will be displayed: CND event 2. After this, or if the destination does not have caller display, the call progresses as normal.

Figure 4 shows the template for CFBL (Call Forward on Busy Line). It allows a phone that is busy to have calls forwarded to another number. The feature is activated whenever a number is dialled. Forwarding occurs if the number being dialled (*Q*) is busy but has a number for forwarding on busy (*ForwardBusy Q* is determined). If so, the dialled number is changed to the forwarding number. Alternatively, no action is taken if the called number is idle or does not have forwarding on busy. In this case the call progresses as normal: the destination is rung if it is idle or the caller gets busy tone if the destination is busy.

A forwarded call is recorded in the *Via* status variable. If a call from number 1 to number 2 is forwarded to number 3, *Via 1 3* records *2* as the number via which the call was made. Since a call may be forwarded to another busy number, several redirections may occur. This is why CFBL loops back to the start, meaning just after where the feature was triggered. This loop is discussed again in section 2.4. The new forwarding number is checked for being busy, and may lead to further forwarding. (In practice, telephone networks enforce a limit on the number of forwarding attempts.)

Sometimes it is not desirable to apply a template. For example, CFBL is meant to be inserted anywhere an actual phone number is dialled. However some uses of dialling are not to establish a call. INTL in figure 2 contains an example (event 7) where dialling is used to enter a PIN. It would be inappropriate to insert CFBL after such a triggering event. For this reason, template matching can be suppressed by placing '!' ('no match') after the node number.

2.4 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 network behaviour (for the operator

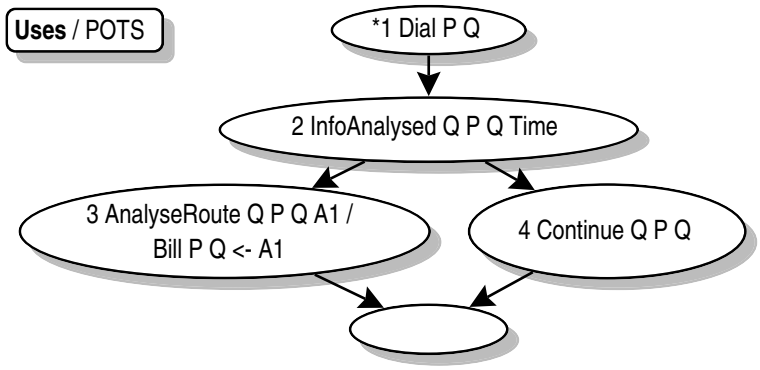

Figure 5: CRESS Feature Diagram for IN Freephone Billing

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 at local rates and the callee pays the rest.

Figure 5 shows the feature INFB (IN Freephone Billing) as an example of how CRESS handles billing. This features causes the destination to pay for a call. Dialling triggers a request to the SCP to determine who should pay for a call: INFB event 2. If this is other than the caller (*P*), the SCP replies with the paying address *A1*: INFB event 3. For freephone this would be the callee (*Q*), but for other billing arrangements could indicate the split of billing. INFB records the paying party in the *Bill* status variable. If no special billing arrangements apply, the SCP asks the call to continue as normal: INFB event 4. By default, the caller is charged for a call.

Now consider the loop in figure 4 again. A complication arises if several features are triggered by the same event. For example INFB, INCF (IN Call Forwarding) and INFR (IN Freephone Routing) are each enabled by dialling as well as CFBL. Their instances are appended in sequence to the *Dial* node. A loop back to the trigger of a feature template is actually to the start of such a feature chain.

INFB first checks which party will pay the bill. INCF, INFR and CFBL then have the opportunity to forward the call (unconditional for INCF, controlled by source and time for INFR, depending on destination busy for CFBL). If the call is forwarded then the feature chain is invoked again, starting with INFB. This is necessary because the new destination may have different charging arrangements. By the time the call reaches its destination, it may have been forwarded several times. The billing for each call redirection will also be different.

The simple *LogBegin/End* events used by CHISEL are therefore insufficient. Although these are allowed by CRESS, the *Start Billing* and *Stop Billing* events should be used; they appeared in nodes 6, 8 and 11 of figure 1. In fact, these are macro events that expand to primitive *LogBegin/End* events for each redirection. The stored *Via* and *Bill* values are used to infer the redirections and the paying parties.

Dialling is a major nexus in telephony for features to be invoked. As well as those just mentioned, OCS/TCS (Outgoing/Terminating Call Screening) are also triggered on dialling to forbid calls to/from certain numbers. Fortunately the feature composition mechanism automatically handles the chaining of all these features. The designer can describe each feature in isolation (the goal of modularity), and their combination is automatic. In fact there are certain

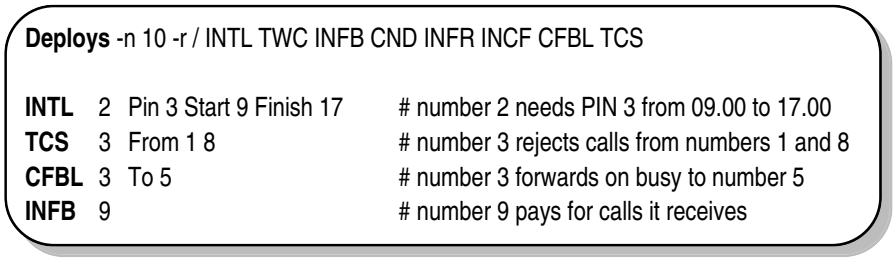

Figure 6: CRESS Configuration Diagram

precedence rules that have to be enforced. For example billing must be considered before call forwarding, and terminating call screening must be applied to the final number obtained after forwarding. The tools ensure that features are combined in a sensible order.

2.5 Configuration

A special configuration diagram is used to define the features and the user profiles. A small example is shown in figure 6. The **Deploys** statement has the form: *Options / Features*. The options are translation parameters. Here, they ask that 10 call instances be generated (*-n 10*) and that call instance behaviour be repeated on hang-up (*-r*). POTS is implicit in the list of features but can be given explicitly if preferred. User profiles appear after the **Deploys** statement, giving the features and their parameters selected by each user.

3 Tool Support

This section gives an overview of the CRESS tools, with particular reference to how LOTOS is generated and analysed.

3.1 Toolset Structure

Figure 7 shows the relationship among the CRESS tools. Symbols are shown doubled where there may be several files or several variants of a tool. The boxed area in figure 7 is the CRESS toolset. Outside this, the diagram editor and 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-known formats. CRESS is thus not dependent on a particular diagram editor.

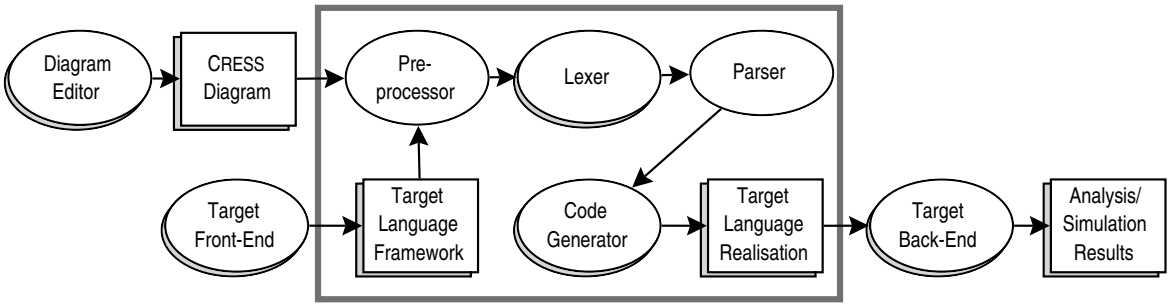

Figure 7: CRESS Toolset

CRESS is also not bound to any particular target language. Currently 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 a very early stage, so E-LOTOS is not yet a target for CRESS. The SDL translator does not generate advanced SDL constructs, and is in fact compatible with SDL 88. Later versions of SDL such as SDL 92 were considered, but CRESS does not really gain anything from them. Other target languages have been thought about, including translation to Java (Beans). The choice of target language depends on the intended use of a CRESS-generated specification. LOTOS offers good analytic capabilities, SDL is industrially attractive, while Java (Beans) would provide an interesting development route.

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 telephony frameworks for LOTOS and SDL are currently available. The framework provides the architecture in which the features are embedded. For example, in telephony the framework defines the behaviour of the SCP, status manager and billing system.

3.2 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 (LOTOS) or button click (SDL) can activate the CRESS toolset automatically. The CRESS pre-processor scans the target language framework for CRESS macro calls:

Cress(Types)	(* generate domain-dependent data types *)
Cress(Profiles)	(* generate user profile information *)
Cress(Features)	(* combine root diagram and features *)

Each of these is expanded to the corresponding definitions in the target language. The data types 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 data types.

Each CRESS macro 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 features 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.

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 also 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.

Specifications are generated with acceptable speed. For example, the configuration shown in figure 6 is translated from diagrams to LOTOS in about 5 seconds on a Pentium 450. The resulting specification is about 1800 non-comment lines of LOTOS. In fact the translators can optionally produce very detailed comments on the generated code.

3.3 LOTOS as A Target Language

3.3.1 Translation to LOTOS

As an example of how CRESS is translated, the strategy for generating LOTOS will be described. (The translation to SDL is broadly similar, but has to deal with some nasty complications due to SDL restrictions.) The outline structure of the generated specification is given below. Processes communicate via the gates *Bill* (log billing events), *Scp* (signals to/from the SCP), *Stat* (read/write status variables and profiles) and *User* (signals to/from the subscribers). Of these just the *User* gate is externally visible, so only events like going off-hook or the phone ringing can be seen. The *CallInstances* process represents all concurrent calls. The *ServiceControl* process defines the SCP. Status variables and profile information are managed globally for all calls by the *StatusManager*. Finally, the *BillingSystem* process simply accepts billing events. Calculation of call charges is presumed to be specified separately.

```
Specification Network [User] : NoExit                          (* network *)
  Library ...                                                  (* library types *)
  Type ...                                                     (* pre-defined data types *)
  Type ...                                                     (* domain-specific data types *)
  Behaviour                                                    (* overall behaviour *)
    Hide Bill,Stat,Scp In                                      (* hide internal signals *)
      (
        (
          CallInstances [Bill,Scp,Stat,User]                   (* call instances *)
        |[Scp]|                                                (* synchronised on SCP messages *)
          ServiceControl [Scp,Stat]                            (* service control point *)
        )
      |[Stat]|                                                 (* synchronised on status messages *)
        StatusManager [Bill,Stat] (...)                        (* status manager *)
      )
    |[Bill]|                                                   (* synchronised on billing messages *)
      BillingSystem [Bill]                                     (* billing system *)
Where
  Process ...                                                  (* processes *)
```

CRESS Construct	LOTOS Construct
diagram parameter	process parameter
Any	dummy variable (input) or *AnyAddress* (output, expression)
node	direct translation (once-used node) or process call (revisited node)
node contents	nothing (empty node) or events and assignments (non-empty node)
event ||| event	';' (no concurrency) or '|||' (concurrency)
event	event offer
event parameter	'!' (known variable) or '?' (unknown variable)
assignment	set (status) variable value from expression
variable use	read status manager (global variable) or direct use (diagram variable)
variable update	write status manager (global variable) or **Let** (diagram variable)
Time	read status manager clock
leaf node	followed by **Stop** (no repeat) or recursive process call (repeat)

Figure 8: CRESS to LOTOS Mapping

Each call instance deals with an arbitrary pair of subscribers; the actual phone numbers are fixed only during the call by going off-hook and dialling. The behaviour of a call is determined by the root diagram and features in use. The number of call instances is defined as a translator option. Although CRESS can handle an arbitrary number of concurrent calls, too many complicates the analysis. In practice, three concurrent calls are sufficient to discover most interesting feature interactions. A translator option decides whether reaching a leaf node (hanging up) terminates a call instance or causes call behaviour to repeat.

Figure 8 summarises the key aspects of the translation to LOTOS. This table hints at a number of complications. Normally a node is translated directly. However if more than one path in the graph converges on the node, then it and its successors are placed in a process definition. References to the node then become calls of this process. The translators have an option to serialise concurrent events. This is because the concurrency adds little to the expressive power of CRESS but greatly complicates analysis. The translators perform a data flow analysis of a graph in order to determine which variables are known at each event. This is partly to detect errors, but it is also necessary to decide whether an event parameter is input or output. Statuses and profiles are held globally for all features by the status manager, so reading or writing a variable requires communication with it. Diagram variables are simpler because they are local and can be accessed directly.

3.4 Feature Analysis with LOTOS

The emphasis in CRESS so far has been on the representation and formalisation of features. Work on feature interaction detection has been limited to date. The following presents preliminary work, but in fact techniques developed by others can be adapted easily with the specifications generated by CRESS.

To check the correctness of features, each was simulated on its own when combined with POTS. Human judgment was used in deciding significant execution paths. The aim was to execute each path at least once. This procedure built confidence in the feature descriptions as well as creating a collection of validation scenarios.

For simple features the number of interesting paths is small. Thus CND (Calling Number Delivery, figure 3) has two such paths: the destination does or does not have caller display. As an example, the following shows a call from phone number 1 to phone number 3 (which has CND and so sees the caller's number):

```
User !OffHook !1                          (* 1 picks up *)
User !DialTone !1                         (* 1 receives dial tone *)
User !Dial !1 !3                          (* 1 dials 3 *)
User !StartRinging !3 !1                  (* 3 starts ringing from 1 *)
User !StartAudibleRinging !1 !3           (* 1 starts ringing for 3 *)
User !Display !3 !1                       (* 3 sees 1 as the caller *)
User !Answer !3                           (* 3 answers the call *)
User !StopRinging !3 !1                   (* 3 stops ringing from 1 *)
User !StopAudibleRinging !1 !3            (* 1 stops ringing for 3 *)
User !OnHook !3                           (* 3 hangs up *)
User !Disconnect !1 !3                    (* 1 is told 3 has hung up *)
User !OnHook !1                           (* 1 hangs up *)
```

For complex features the number of interesting paths is just manageable. TWC (Three-Way Calling), for example, has 23 important paths. Either user may start a three-way call, and there are a number of ways in which three-way setup may fail or may later revert to a two-way call. As a sample execution, the following starts with a call from phone number 1 to phone number 2. A three-way call is then initiated by 2 doing flash-hook and calling phone number 3. However before 3 can answer, 2 does flash-hook to cancel the three-way call and then hangs up. The call from 2 to 3 is cancelled, and 1 is forced to hang up:

```
User !OffHook !1                          (* 1 picks up *)
User !DialTone !1                         (* 1 receives dial tone *)
User !Dial !1 !2                          (* 1 dials 2 *)
User !StartRinging !2 !1                  (* 2 starts ringing from 1 *)
User !StartAudibleRinging !1 !2           (* 1 starts ringing for 2 *)
User !Answer !2                           (* 2 answers the call *)
User !StopRinging !2 !1                   (* 2 stops ringing from 1 *)
User !StopAudibleRinging !1 !2            (* 1 stops ringing for 2 *)
User !Flash !2                            (* 2 does flash-hook *)
User !DialTone !2                         (* 2 receives dial tone *)
User !Dial !2 !3                          (* 2 dials 3 *)
User !StartRinging !3 !2                  (* 3 starts ringing from 2 *)
User !StartAudibleRinging !2 !3           (* 2 starts ringing for 3 *)
User !Flash !2                            (* 2 does flash-hook *)
User !OnHook !2                           (* 2 hangs up *)
User !StopRinging !3 !2                   (* 3 stops ringing from 2 *)
User !StopAudibleRinging !2 !3            (* 2 stops ringing for 3 *)
User !Disconnect !1 !2                    (* 1 is told 2 has hung up *)
User !OnHook !1                           (* 1 hangs up *)
```

Validation scenarios like these characterise the expected behaviour of features of isolation. The scenarios can then be encoded using the ANTEST (ANISE Test) language developed for ANISE [13]. 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 feature being present for a phone number. In fact, ANTEST is used to encode more comprehensive use-case scenarios that synthesise the individual executions obtained through simulation.

The ANTEST tool automatically translates validation scenarios into LOTOS and runs them in parallel with a specification using the *TestExpand* function of LOLA. When features are combined individually with POTS, this merely confirms the validity of the scenarios. More importantly, the scenarios can be run against POTS combined with *all* features. A common interpretation of feature interaction is that a feature fails to perform as expected when other features are present. The manifestation of feature interaction when using ANTEST is either deadlock or non-determinism. Deadlock means that one feature prevented another from working; for example, CNDB (Calling Number Delivery Blocking) prevents CND from working – a desired interaction! Non-determinism means that an ambiguity arose; for example, flash-hook is used by both CW and TWC.

The University of Ottawa have in fact worked on the same features [5] as used here with CRESS, though their LOTOS specifications were created by hand [4]. Their techniques for detecting interactions are therefore also applicable to CRESS. For example non-intrusive observer processes can be used to check for violation of feature properties, and watchdog processes can be used to check for violation of system properties.

In future work, the author intends to apply techniques he developed from protocol conformance testing [9] to derive use-case scenarios automatically. Another avenue that the author intends to explore is the use of symbolic model-checking to verify that feature properties are preserved in the presence of other features. The current validation approach can check that calls to a specific number are rejected if they are on its given screening list. Model-checking should allow such properties to be proved in general.

4 Conclusion

It has been seen that CRESS is a graphical language for specifying systems with a base functionality and additional features. The elements of the notation have been introduced for root diagrams, feature diagrams, rules and configurations. The particular contribution of CRESS is its ability to describe and combine features in a flexible and automatic manner. A portable toolset enables thorough checking and translation of diagrams to various languages, currently LOTOS and SDL. Early work has been presented on feature interaction detection using the specifications generated by CRESS. Future developments will include support for a wider range of diagram editors and target languages. More complete interaction detection techniques will also be developed. Although CRESS has been illustrated on traditional telephony features, it is applicable to a number of problem domains. One immediate new application to be pursued is Internet telephony using SIP (Session Initiation Protocol [7]). It would also be interesting to study non-telephony applications such as word processor design.

References

[1] A. V. Aho, S. Gallagher, N. D. Griffeth, C. R. Schell, and D. F. Swayne. SCF3/Sculptor with Chisel: Requirements engineering for communications services. In K. Kimbler and W. Bouma, editors, *Proc. 5th. Feature Interactions in Telecommunications and Software Systems*, pages 45–63. IOS Press, Amsterdam, Netherlands, Sept. 1998.

[2] D. Amyott, L. Charfi, N. Gorse, and T. Gray. Feature description and feature interaction analysis with use case maps and LOTOS. In M. H. Calder and E. H. Magill, editors, *Proc. 6th. Feature Interactions in Telecommunications and Software Systems*, pages 274–289, Amsterdam, Netherlands, May 2000. IOS Press.

[3] E. J. Cameron, N. D. Griffeth, Y.-J. Lin, M. E. Nilson, W. K. Schnure, and H. Velthuijsen. A feature-interaction benchmark for IN and beyond. *IEEE Communications Magazine*, pages 64–69, Mar. 1993.

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

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

[7] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg, editors. *SIP: Session Initiation Protocol*. RFC 2543 *bis*. The Internet Society, New York, USA, July 2000.

[8] ITU. *Intelligent Network – Global Functional Plane for Intelligent Network Capability Set 2*. ITU-T Q.1223. International Telecommunications Union, Geneva, Switzerland, 1997.

[9] 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.

[10] F. J. Lin and Y.-J. Lin. A building block approach to detecting and resolving feature interactions. In L. G. Bouma and H. Velthuijsen, editors, *Proc. 2nd. International Workshop on Feature Interactions in Telecommunications Systems and Software Systems*, pages 86–119. IOS Press, Amsterdam, Netherlands, 1994.

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

[12] K. J. Turner. An architectural description of intelligent network features and their interactions. *Computer Networks*, 30(15):1389–1419, Sept. 1998.

[13] 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.

[14] 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.

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