Kenneth J. Turner. Formalising the Chisel Feature Notation. In Muffy H. Calder and Evan H. Magill, editors, Proc. Feature Interactions in Telecommunication Networks VI, pages 241-256, IOS Press Amsterdam, May 2000.

Formalising the Chisel Feature Notation

Kenneth J. Turner

Computing Science and Mathematics, University of Stirling
kjt@cs.stir.ac.uk

Abstract.
The CHISEL notation was developed by Bellcore as an informal graphical notation for describing telecomms services and features. CRESS (CHISEL Representation Employing Systematic Specification) is an enhanced version of CHISEL with tightly defined rules for the syntax and static semantics of diagrams. More importantly, CRESS has formal denotations given by SDL (Specification and Description Language) and LOTOS (Language Of Temporal Ordering Specification). This permits rigorous checking, analysis and prototyping of descriptions. The accompanying toolset has been written in an open and extensible manner.

1 Introduction

CHISEL [1] is a graphical language for describing telecomms services and features. It was developed at Bellcore (now Telcordia Technologies) to support the service creation process. Although intended as a rigorous approach, CHISEL presents descriptions in an accessible manner. In particular, it allows stake-holders in service creation to understand feature design without becoming proficient in a formal notation. At its simplest, CHISEL merely describes the sequences of events that characterise a feature. Yet its use in the first feature interaction detection contest [4] demonstrated that it is capable of describing a wide variety of features. The community lacks a common notation for defining features; CHISEL has the potential to fill this role.

CHISEL was initially defined as a way of giving the event sequences that characterise features. A complementary notation was developed to describe sequences of interactions among AIN (Advanced Intelligent Network) components. CHISEL is supported by the Sculptor tool developed at Bellcore. The CHISEL designers have outlined strategies for translating CHISEL diagrams into MSCs (Message Sequence Charts), hierarchical textual descriptions, finite state automata, regular expressions, and basic process algebra.

However, the diagrams used in the feature interaction contest contain new constructs that do not appear to have been part of the original CHISEL notation. In particular, the ability to define separate feature diagrams is powerful but more complex. Unfortunately, the interpretation of these more advanced diagrams is not always clear. The rules for drawing CHISEL diagrams have not been formalised to the author's knowledge. The diagrams are often supplemented by informal commentary that is not rigorously integrated into the descriptions. Certain aspects of the CHISEL notation lead to unnecessary repetition and to some obscurity. Sculptor is proprietary and therefore not publically available; its availability on a variety of platforms is also restricted.

For these reasons, the author set out to extend the CHISEL notation for greater usability, while retaining backwards compatibility with existing diagrams. Diagrams are given formal denotations in two popular formal languages – SDL (Specification Description Language [6]) and LOTOS (Language Of Temporal Ordering Specification [5]). The end result of this work is an improved language called CRESS (CHISEL Representation Employing Structured Specifications) for graphical feature description, analysis and prototyping. CRESS is supported by a toolset than runs on many different platforms and can be used with a wide variety of target languages.

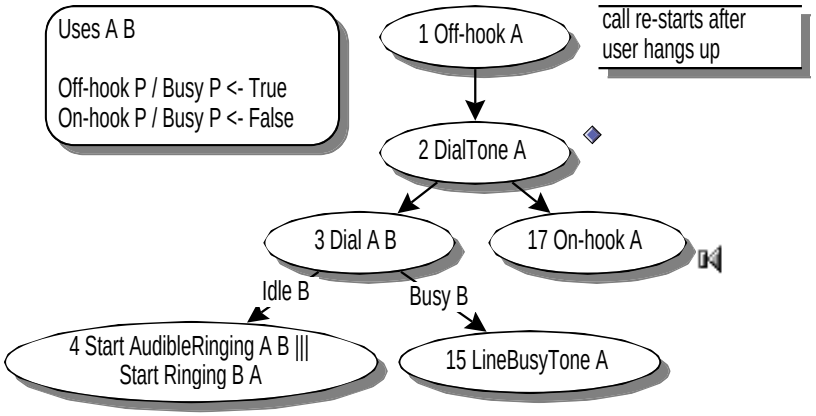

Figure 1: A Partial Root Diagram for POTS (Plain Old Telephone Service)

A number of authors have adopted architectural (structured) approaches to feature description (e.g. [2, 7, 10]). The author himself has developed a separate approach called ANISE (Architectural Notions In Service Engineering, e.g. [9]). The goal of such approaches is to have a well-defined architecture that supports creation, specification, analysis and development of features. The work reported in [3] is directly comparable to the LOTOS translation undertaken by CRESS, but makes use of hand-crafted LOTOS specifications.

2 The CRESS Notation

2.1 Basic CRESS Diagram Concepts

CRESS extends the CHISEL vocabulary for describing diagrams. A CRESS diagram defines the behaviour of a system – a switch, another network component like an SCP (Service Control Point), or the network as a whole. CHISEL uses the term 'platform' to mean the participants in a feature and the rules for the signals they exchange. CHISEL diagrams like those in [4] often describe a user view, treating the network as a black box.

Figure 1 shows part of a root diagram that describes a self-contained service, here POTS. Generally speaking a root diagram defines a base telephony service. However note that POTS is *defined* in CRESS; it is not built-in, as with the IN and similar approaches. This makes CRESS more widely applicable, e.g. for mobile communication services or multimedia services.

In general a CRESS diagram is a directed cyclic graph. A diagram has numbered event nodes like *1* and *2* in figure 1. The shape of an event node is unimportant; shadowed ovals are used here, while ovals and rectangles have been used for CHISEL. A node contains input or output signals (but not both) such as *Off-hook* and *DialTone*. Signals carry parameters that are often the addresses of the participants (their telephone numbers). Several signals in a node may be processed independently in parallel (e.g. node 4 in figure 1).

Event nodes are linked by arcs to show the flow of control. An arc may be labelled with a boolean condition as a guard on the occurrence of a transition (e.g. *Busy B* in figure 1). When guards become complex (e.g. see Three-Way Calling in [4]), there is a risk of giving inconsistent guards (they may not be disjoint, and may not amount to a tautology). To reduce the risk of error, CRESS allows one of the guards leaving a node to be labelled **Else** (meaning the negated disjunction of all other guards).

It is often difficult to persuade developers to give adequate commentary on their designs.

CRESS makes this easier by providing several mechanisms to add comments easily. The most conventional idea is a comment box, shown beside node 1 of figure 1. As a second mechanism, CRESS supports attachments. These are files that the user can open by clicking on the associated marker (the small diamond next to node 2 in figure 1). Attachments can be any kind of file including other CRESS diagrams. Perhaps the most convenient kind of attachment is a sound annotation (the small loudspeaker next to node 17 in figure 1). Since developers can be reluctant to write full comments, they are encouraged to record a verbal explanation as a spontaneous note of their thoughts. Clicking on a sound attachment replays the comments made while the diagram was being developed.

The greatest informality in CHISEL stems from the textual description of how call status variables and feature parameters are manipulated. For example, [4] gives informal rules in this way. Apart from the possible ambiguities of natural language, such rules are not integrated into the notation and therefore cannot be enforced by tools. CRESS addresses this problem by supporting rule boxes (the rounded rectangle in figure 1). A rule box contains a *Uses* statement, and may also define signal assignments, function definitions and variable initialisations.

A *Uses* statement declares the feature parameters (generic subscribers *A* and *B* in figure 1) and any subsidiary diagrams. A more complex example would be *Uses A B C / CND POTS*, where '/' separates parameters from diagrams. The first part optionally gives the feature parameters (here *A*, *B* and *C*). A feature that builds on another might not introduce new parameters, so this part of the statement may be empty. Feature parameters accumulate as features are combined; they are used for statically checking diagrams and during code generation. The optional second part of *Uses* names the diagrams that the current diagram depends on. In the above example, the feature depends on CND (Calling Number Delivery) and POTS. CRESS incorporates subsidiary diagrams automatically, handling multiple references and even self-references.

Most of the informal rules in CHISEL describe how call status variables change as a result of signals occurring. Such rules can be written explicitly into event nodes, separating signals from assignments by '/'. For example, node 4 in figure 1 might be written out in full as:

```
    StartAudibleRinging A B / AudibleRinging A B <– True
|||
    StartRinging B A / Busy B <– True   Ringing B A <– True
```

Call status variables like *Ringing* are usually parameterised by addresses. The '<–' symbol denotes assignment of an expression. If a signal is followed by several assignments, these may be syntactically ambiguous. (More exactly, the CHISEL grammar is context-sensitive rather than context-free and is thus trickier to parse.) In such cases, CRESS requires the use of a '/' symbol between each ambiguous assignment. This is good practice anyway as it helps readability.

Although diagrams can be drawn with explicit assignments like the above, they quickly become tedious to create and to read. In fact, signal assignments can largely be captured by simple rules. A CRESS rule box allows signal assignments to be defined. Figure 1 shows two of the rules governing whether a subscriber is busy. Parameters like *P* are generic, and are replaced by the actual parameters of a signal. The use of such rules greatly simplifies the descriptions of event nodes. However in some cases, e.g. busy for Call Waiting or Three-Way Calling, the rules are irregular and cannot be so easily captured. It is therefore possible to give exceptions to such rules directly in event nodes; explicit assignments override those implied by the rules.

A rule box may also contain function definitions. For example, a line being idle is defined as *Idle P* $\leftarrow \sim$*Busy P*. This format of rule is distinguished by not being prefixed with a signal. In fact signal assignments and function definitions are handled by a macro processor in the CRESS

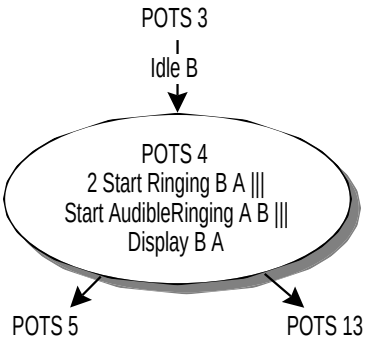

Figure 2: A Feature Diagram for CND (Calling Number Delivery)

parser. Arbitrary function definitions like *Markup Cost Percent* $\leftarrow$ *Cost * (1 + Percent / 100)* are supported in this way. The macro processor is also used for certain internal functions within the CRESS tools. Macros can be used to simplify the appearance of complex guards and parameters in the graphical part of a diagram.

There does not appear to be a definitive description of CHISEL expressions. The operators permitted by CRESS are as follows (in decreasing order of binding precedence):

$-$ $\sim$	* / %	+ $-$ **Not In**
= != < <= >= > **In**	&&	|| ^^

These include set membership and its negation (**In**, **Not In**) and exclusive or (^^). Expressions may also use parentheses, **If**...**Then**...**Else**...**Fi**, **Any** (any subscriber), indexed variables (like *Ringing B A*) and **Time** (the current clock).

Finally, a rule box may give a variable initialisation like *Integer PeakRate* := *4*. Permissible variable types are *Address* (subscriber number), *Boolean*, *Cadence* (special ring tone), *Integer*, *Message* (character string), *PIN* (Personal Identification Number) and *Time*. Initialisations are performed before the first node of a diagram.

CRESS diagrams may contain loops. As a result, this can lead to ambiguity about what the first node of a diagram is. In such cases, a node marked **Start** must be added as the top-level node. In fact such a node is always implied if it is not given explicitly. CRESS also supports the CHISEL notion of a **NoEvent** node that performs no action. It is occasionally useful where *m* nodes have to be connected to *n* nodes. Instead of $m \times n$ arrows between all pairs, they can be connected via an intermediate **NoEvent** node.

2.2 Advanced CRESS Diagram Concepts

Features are generally regarded as modifying a base telephony service in some way, though they may also be free-standing. For IN-like features and those appearing in [4], the descriptions are given as the changes to POTS. A feature diagram shows how another diagram is changed by addition, deletion and modification of nodes (and guards). Features may modify a root diagram or another feature diagram. Figure 2 shows the feature CND (Calling Number Delivery). For a number of features such as this, the CRESS diagrams are simpler than the ones in [4].

A CRESS (or CHISEL) feature diagram is modular in the sense that it defines a feature separately. Like any module it has interfaces – the elements of the root diagram that it links to. However a feature does not exhibit a strong semantic modularity. Although a feature diagram can be considered on its own, it needs to be seen in the context of the root diagram and is thus not completely independent. Similarly, features in the IN are invoked at various points in call and

return to other points of the call. From an object-oriented viewpoint, a feature specialises a base description by 'inheriting' behaviour and modifying it. There may be a behavioural subtyping relationship for some feature combinations (e.g. POTS+TCS $<$ POTS) and not others (e.g. POTS $\not<$ POTS+CFU). Despite attempts like [8], there is little real opportunity for true object orientation in typical telecomms services.

The first node in a feature diagram is termed a source node; it locates the diagram node that is about to be changed (in figure 2, POTS node 3). This is followed by nodes that add to or replace other nodes in the original diagram. The nodes of a feature diagram are numbered independently; in fact the node numbers for a diagram are implicitly prefixed by the diagram name. CHISEL does not appear to have a notation that allows a new node to be appended to the first node of a root diagram. CRESS allows addition to an initial node such as **Start** in POTS.

A feature may simply add new nodes not present in the original diagram. The arcs leading to these nodes may have guards that are additional to the original. A feature may also replace nodes of the original diagram. In this case, a source node is followed by a swap node: a descendant of the source node that is to be replaced. In figure 2, node 4 of POTS is completely replaced by node 2 of CND. Since the original arc between nodes 3 and 4 of POTS is guarded by *Idle B*, the feature diagram is similarly guarded. The static semantics of CRESS requires that this arc in the feature diagram corresponds exactly to that of the original.

A feature diagram may contain leaf nodes as in a root diagram. Most commonly, a feature diagram continues with other nodes in the original diagram; these destinations in a feature diagram are called sink nodes. In figure 2, node 2 is followed by sink nodes 5 and 13 of POTS. The effect of a feature diagram is therefore to splice a new graph into the original. In doing so it may augment, alter or delete parts of the original.

CHISEL diagrams show source, swap and sink nodes with complete bindings of all feature parameters, e.g. *POTS A<–A B<–B 3*. As study of [4] will show, a great many source, swap and sink nodes contain uninteresting bindings like this. To simplify diagrams, CRESS allows identity bindings to be omitted; indeed it suppresses them if they are given. The example above is therefore simplified to *POTS 3*. A diagram with loops may have sink nodes in the same diagram. The diagram name in a sink node can be omitted, meaning the current diagram. Thus '2' would mean node 2 in figure 2 as a destination.

For sink nodes, the CRESS interpretation of bindings is as expected: make the parameter substitutions on moving to the new destination. For source and swap nodes, CRESS interprets the bindings *backwards*. Thus *A*$\leftarrow$*X* means '*A* corresponds to *X* in the feature diagram'. If the binding were interpreted as 'substitute *X* for *A*', it would alter all uses of *A* in the root diagram. This would interfere with other features modifying the root diagram. Instead a source or swap binding modifies the *feature* diagram, allowing a number of features to be combined.

Like CHISEL, CRESS allows root diagrams or feature diagrams to be split into pieces (typically to give page-sized chunks of description). The intra-diagram connectors are called arrows (to) and targets (from) in CRESS. By convention such connectors are labelled alphabetically (e.g. *TWC D*), thus distinguishing them from the numeric labels used in ordinary nodes. Although arrows/targets are similar to sinks/sources, they may not give parameter bindings. Targets may also be qualified by a guard that controls their applicability (e.g. see the Return Call feature in [4]). Arrows/targets connect nodes strictly within the same diagram. As might be expected, the diagram name is normally omitted for an arrow or target (e.g. a simple reference like *D*).

Loops in a diagram can be drawn explicitly. They can also arise through use of sink and arrow nodes. Figure 3 shows *TESTR* (Test Repeats) – a fairly pathological diagram with

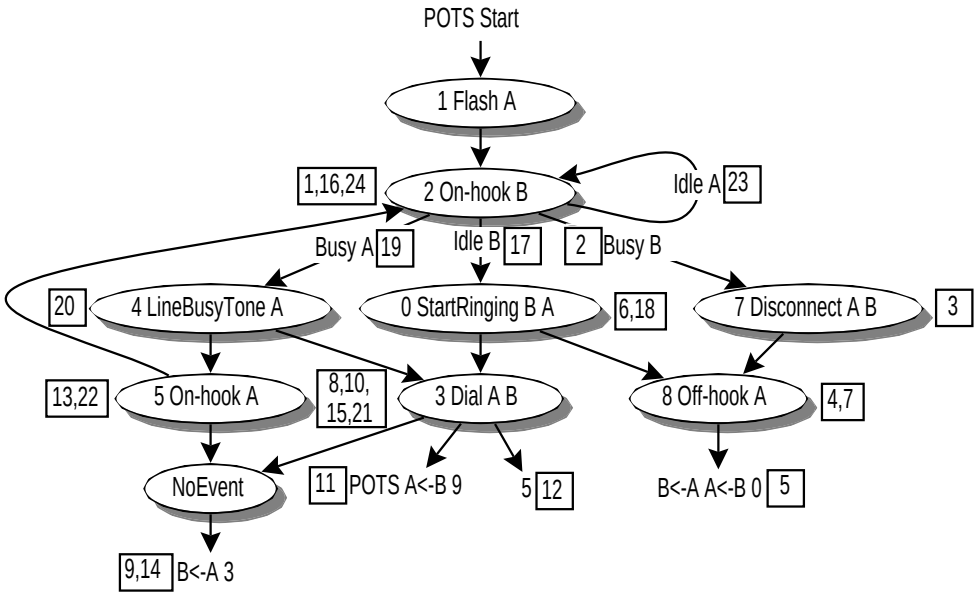

Figure 3: *TESTR* (Test Repeats): A Pathological Feature Diagram

complex direct and indirect loops. The numbers in boxes are not part of the CRESS notation; they are used later to explain code generation. *TESTR* makes no sense in telephony terms, and is given here only as an indication of diagram complexity. In fact it is part of the CRESS tool regression test suite. Its translation to SDL and LOTOS appears in sections 4 and 5.

2.3 Relationship between CRESS and CHISEL

CRESS is a superset of CHISEL, so any CHISEL diagram (subject to disambiguation of syntax) can be processed by the CRESS tools. However CRESS offers a number of simplifications and more tightly defined rules. The grammar of CRESS diagrams has in fact been formulated using the metasyntax used for SDL diagrams [6]. A list of the differences between CHISEL and CRESS is available from the author. Most of the syntactic and static semantic constraints on CRESS diagrams are fairly obvious, but a number of non-trivial grammar rules are also enforced.

The author has re-drawn diagrams from [4], taking advantage of the improvements offered by CRESS. The CRESS diagrams are not identical to those of [4], partly due to the simplifications but more importantly because automated analysis found a number of technical errors in the original CHISEL diagrams. Some of these errors are simple (but easily missed) mistakes. The more serious errors concern the logic of the features. It seems that the diagrams of [4] have only been hand-drawn and not checked with tools like those about to be described.

3 Tool Support for CRESS

3.1 Toolset Architecture

As a graphical notation, CRESS is just a drawing aid and has no formal semantics. However the interpretation of CRESS corresponds closely to an LTS (Labelled Transition System). Rather than re-invent the wheel by defining semantics using an LTS and building tools to support this, it is preferable to give the semantics through an existing LTS language. Both SDL and LOTOS (at least in their bare forms) have LTS semantics and rich tools. Since the basics of CRESS are

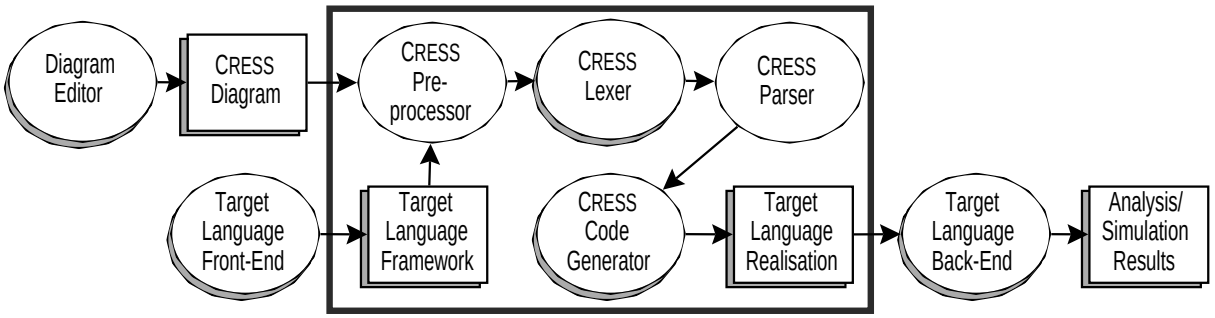

Figure 4: The CRESS Toolset Architecture

rather simple, there can be confidence that its denotations in SDL and LOTOS are compatible. Denotations in other (operational, constructive) formal languages would also be possible.

CRESS is supported by a set of tools for parsing, checking and translating diagrams. The overall tool architecture is illustrated in figure 4. The symbols shown doubled indicate where a number of instances may occur. The diagram editor and target language tools are external to CRESS and not part of its toolset.

The main tools work from the command line, and can be run this way by the user. However they are normally invoked automatically. The designer creates a set of diagrams using whatever graphical editor is convenient. The target language tools are then invoked for one of the framework specifications. The framework is fixed and independent of the individual features, but is specific to the target language. Most target languages support preprocessing of their input. The CRESS preprocessor expands the framework specification and generates code from the named diagrams. These are incorporated into the final realisation that is then analysed or run as a prototype. As indicated by the grey rectangle in figure 4, the user sees only the CRESS diagrams and the resulting analysis or simulation. A simple invocation of the target language tools (e.g. a button click in Telelogic SDT) carries out the translation and analysis. The use of any particular target language is thus largely invisible to the user.

Any reasonable diagram editor can be used. The author uses Lighthouse Design's *Diagram!* that runs on five different platforms. *Diagram!* is ideal for drawing CRESS diagrams, and can be tailored for the application domain. For example, the author has created a palette of the symbols used in CRESS diagrams. It is then a simple matter to drag the selected symbol on the drawing area. *Diagram!* also supports the notion of arcs directly connecting symbols (unlike a number of diagram editors where symbols and arcs are separately drawn). The file format used by *Diagram!* is already readily parsed. 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. However a different version of the CRESS lexer (lexical analyser) is needed for each diagram format. Fortunately the lexer is a fairly straightforward and small part of the toolset. Much of the code for *Diagram!* could be re-used.

There is also freedom in the choice of target language. Since the author is interested in formal analysis, SDL and LOTOS are alternative targets. SDL is the industry-standard language in telecomms and an obvious choice. The SDL code generated from CRESS is compact and human-readable, and so may even be of use in product development. LOTOS offers better analytic capabilities, and is the preferred choice for verification. Translation to conventional languages like C(++) or Java should also be quite feasible. SDL is sufficiently close to a programming language that its code generator provides evidence of this claim. Note that the current target

languages supported by CRESS complement the capabilities of CHISEL and Sculptor.

In developing tool support for CRESS, the author has specifically had openness and portability in mind. The freedom to choose diagram editor and target language are two factors. The tools have been written using Perl (Version 5) and will thus run on all major platforms. The code has been documented in great detail, helping others to use it and to make adaptations for other diagram formats and target languages.

Since the CRESS tools essentially perform the task of a compiler (indeed they are structured as a normal compiler would be), it might have been expected that a conventional *lex/yacc* approach would have been used. In fact, this would have supported only the basic parsing of CRESS diagrams (about 15% of the total code in the toolset). The remainder would have to have been written in C (or possibly C++). Translating CRESS requires very substantial amounts of string handling and pattern-matching, for which the C(++) libraries provide limited support. Perl offers extremely flexible handling of data, and is very suitable for writing the 85% of the toolset that supports the complexities of checking and code generation.

The toolset comprises nine modules, totalling 4200 non-comment lines of rather intricate Perl. The sizes of the key tools are of interest: lexer 289 lines, parser 1266 lines, SDL code generator 1043 lines, LOTOS code generator 936 lines. The lexer is relatively small (and large amounts of the code can be re-used for other diagram formats). The SDL code generator is a little larger than that for LOTOS. However, it will be seen that SDL code generation is much more difficult than for LOTOS.

3.2 Tool Operation

The CRESS preprocessor expands a framework specification as described later for SDL and LOTOS. The preprocessor translates statements of the form *Cress(...)* into the target language. These statements are macro calls that name the CRESS elements to be imported. In fact the *Cress* macro is used exactly twice in a target language framework: to incorporate supporting definitions (mainly types), and to incorporate code for the named diagrams. The preprocessor automatically calls the lexer, parser and code generator (for the target language).

The CRESS lexer identifies nodes and arcs in a diagram and builds a directed graph. This is not as easy as it might seem, since the nodes and arcs may appear in any order in a diagram file. The possibility of cycles within the graph also complicates the procedure. The lexer first extracts all the arcs from the file, and constructs a graph of empty nodes using the adjacency of arc endpoints. Then the lexer fills in the contents of nodes and guards from the diagram file.

The CRESS parser takes the graph read by the lexer and parses all nodes. The graph is checked for syntactic and static semantic correctness. There are an astonishing number of ways to make mistakes in diagrams. The CRESS tools detect 70 error conditions, and make a further 30 sanity checks. A conventional parser builds an abstract syntax tree, but the CRESS parser builds an abstract syntax *graph*. Systematic transformations are performed as the graph is built:

- The syntax inherited from CHISEL is a little inconvenient. Signal names and expressions are therefore normalised. For example, signal *Off-hook* does not conform to usual identifier rules and is changed to *OffHook*. Parameters listed after a signal name or indexed variable are placed in parentheses, e.g. *LogEnd A B Time* is transformed to *LogEnd(A,B,Time)*. Guard expressions are also normalised.
- Parallel inputs or outputs (‘|||’) are split into separate signals. Signals and signal assignments are also separated.

- Arrow nodes are matched to target nodes, and the two are merged. Sink nodes in a feature diagram are matched with the corresponding originals.
- Source and swap nodes in a feature diagram are matched with the originals so that the feature diagram can be spliced in.
- **NoEvent** nodes are eliminated by inserting direct links between the indirectly linked nodes. **Else** guards are placed at the end of guard lists so as to simplify code generation. Several inputs after a node are sorted by signal name to simplify (SDL) code generation.

The end result is a single, possibly cyclic, graph for the root diagram merged with its feature diagrams. The graph contains only input, guard, output and sink nodes. The sink nodes are retained because their bindings are needed during code generation. Finally the whole graph is checked for static correctness.

One of the CRESS code generators now traverses the graph and produces code from it. The graph is traversed depth-first, but because it may be cyclic each node is marked as it is visited. If a node is revisited, the code generator may ignore it or generate code for it; the latter is necessary in some cases. However the descendants of a revisited node are not further traversed. The code generators support common options:

Generate Comments: For human readers, the code generators can automatically annotate their output with comments that explain how the code relates to the CRESS diagram.

Interleave Signals: CRESS allows parallel inputs and outputs. This does not greatly enhance the expressive power of CRESS, although it simplifies the diagrams. By default the code generators serialise any parallel inputs or outputs. This makes the code much simpler and reduces the state space required for verification. If required, the code generators can generate code that interleaves inputs or outputs.

Repeat Behaviour: By default, behaviour terminates at a leaf node; the corresponding call instance dies. If preferred, the whole behaviour can be restarted after a leaf node.

Swap Labels: A swap node is usually handled by preserving the label of the original that is replaced. For example, figure 2 causes the original node to stay labelled as *POTS 4*. This makes it possible for several features to add to a node in the root diagram. However this can be problematic if the features modify the root diagram in inconsistent ways. The code generators can therefore be asked to use the label of the replacing node. In figure 2, for example, this is *CND 2*. A feature modifying *POTS 4* after inclusion of CND would thus have to refer to *CND 2*. This deliberately forces the designer to recognise the interdependency of features, ensuring they are combined in a statically consistent manner.

Table 1 shows the size of code generated *in addition to POTS* for a sampling of the features found in [4]; INCF is IN Call Forwarding. Ottawa University's contest submission [3] is directly comparable and also appears in the table. Since SDL and LOTOS are rather different languages, a comparison in terms of lines of code is not necessarily obvious. However as the layout conventions illustrated in figures 6 and 7 show, the comparison is reasonably fair. Except for data types (for which LOTOS is more verbose), the CRESS LOTOS specifications are a little smaller than than their CRESS SDL counterparts. The CRESS LOTOS specifications also have fewer declarations. The numerous states and joins in SDL lead to spaghetti-like code. Subjectively, the LOTOS specifications are rather easier to read.

The comparison of CRESS-generated LOTOS and the Ottawa hand-generated LOTOS is interesting. The layout conventions are similar in both cases. The Ottawa code is significantly longer with more processes. The CRESS code (with automatically generated comments) is

	CFBL		CND		INCF		POTS			TWC	
	Code	Decs	Code	Decs	Code	Decs	Code	Decs	Defs	Code	Decs
CRESS SDL	62	4	2	0	9	1	94	7	196	506	27
CRESS LOTOS	56	0	2	0	9	0	67	1	218	430	17
Ottawa LOTOS	165	6	41	1	165	6	310	12	964	813	15

Code: lines of behaviour code *Decs*: number of declarations (SDL states, LOTOS processes)
Defs: lines of definitions (SDL data types and signals, LOTOS data types)

Table 1: SDL and LOTOS Statistics for Sample Features

actually better commented than the Ottawa code. The most striking difference is that the POTS specification is much lengthier in the Ottawa approach. This is because the Ottawa group have built a number of feature calls directly into POTS, so this code should really be counted against the feature and not POTS. This would make the Ottawa feature specifications even longer than shown. The Ottawa approach is also less modular in that an integrated POTS specification has been produced. For CRESS, feature specifications are automatically integrated with POTS *as required*. Despite the fact that the CRESS specifications are machine-translated, they compare well with the manually written specifications developed by Ottawa.

4 Supporting CRESS with SDL

4.1 SDL Specification Framework

The target framework for SDL is shown in figure 5. The symbol at the top left of this figure is an SDL (and CRESS) macro call. When the CRESS preprocessor expands this, all the definitions needed for the SDL framework are inserted at this point. As well as data types, *Types* expands to the signal definitions required for communication in SDL.

Telephone subscribers form the environment of the system. The *Switch* process is the key element that executes services and features in a call. The notation '(0,10)' means that zero calls exist initially, and a maximum of ten calls can exist simultaneously. The switch interacts with the *BillingSystem* process to log start and end times for calls between subscribers. The switch also interacts with the *SCP* process to handle IN-like features.

The *StatusManager* process represents the distributed control of lines across the whole network. For example, line busy is handled by this process. Features selected per subscriber are recorded by the status manager or SCP. The status manager routes user input signals to the appropriate switch process instance. This complication arises because in SDL it is the *sender's* responsibility to determine which process instance receives a signal. The subscriber is of course unaware of the internal network operation and cannot do this. The status manager uses its knowledge of line status to direct user signals to the correct switch instance. The act of going off-hook causes the status manager to create a switch process instance (if possible). Subsequent signals from this subscriber are sent to the same instance. The status manager also notes which subscriber is dialled in a call, and routes signals from this subscriber to the correct instance. In features like CW and TWC, more than two subscribers may be associated with a call. Finally, clearing a call breaks the association between subscribers and the switch instance.

The status manager, billing system and SCP processes have fixed definitions in the SDL framework. The switch process definition simply calls the *Cress* macro to include the diagrams to be analysed, e.g. *Cress(CFBL,CND,INCF,TWC)*. POTS need not be named because it is

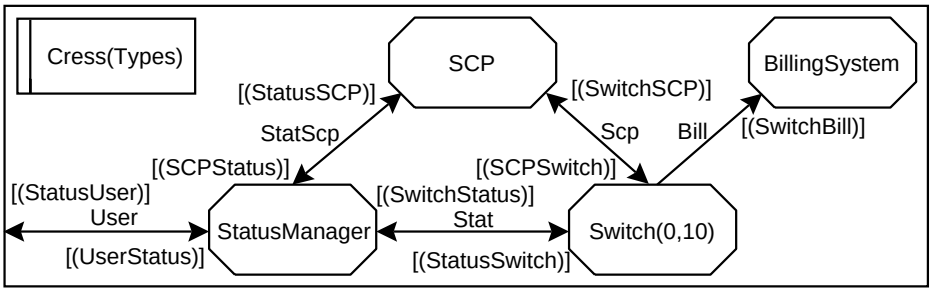

Figure 5: SDL Framework Architecture

included implicitly. The combination of features is up to the designer: either a single feature plus POTS, or any combination of features including all of them. The approach is not restricted to analysing just pairs of features in combination.

4.2 SDL Code Generator

CRESS diagrams are translated to a plain form of SDL. Although a serious attempt was made to use the object-oriented features of SDL 92 (and later), these have subtle restrictions and do not work properly for CRESS. Also, commercial SDL tools do not yet support object orientation fully. For these reasons, the translation uses a simple state machine representation.

The SDL code generator was the first one written for CRESS. It proved to be rather tricky, largely because the rules for inputs in SDL cause immense complications. Inputs in SDL are passive, asynchronous and severely restricted in the syntax. The passive nature of inputs means that a process has no control over the arrival of signals. This is why the status manager has to route subscriber signals to the correct switch instance. The asynchronous nature of inputs means that they are queued and only handled later by the process. If an input is unwanted it has to be discarded; the sender cannot be prevented from sending something undesirable. The syntax for inputs requires them to appear immediately after a state, at the start of a transition.

The SDL code generator therefore has to define a state prior to each input; the node label is used for the state (e.g. *POTS.1* for the off-hook input in figure 1). Since a later part of the diagram may branch to an input node, the transition that follows an input is preceded by a SDL label (also named after the input node).

Any assignments associated with the input then follow. Feature parameters can be used or assigned directly in an SDL task. However, call status variables like line busy need special treatment. These are owned by the status manager process. To access them requires the *View* feature of SDL. The SDL syntax is *View(Busy)(B)* to check the busy status of line *B*. To update a global status variable is more complex. This requires an explicit signal to the status manager, e.g. *Update(Busy,B,True)*. Normally these mechanisms for reading and writing status variables would be unsafe because the variables should be used under mutual exclusion. However the scheduling strategy of a typical SDL toolset can be set for atomic execution of transitions.

A further complication with input is that SDL does not allow the same signal in several inputs leading from a state. Unfortunately this situation is common in CRESS diagrams. The code generator is forced to create one input statement, and then to check which of the expected parameters was received. Suppose that either *A* or *B* may go on-hook. The translation will have a single statement *Input OnHook(In_0)*. (The dummy input parameters are numbered *In_0*, *In_1*, etc.) The subscriber address *In_0* is then checked to see if it is *A*; if not it is assumed to be *B*.

This should be safe since the status manager will send signals from only *A* and *B* to the process instance. For some purposes it is desirable to make an explicit check that input parameters are as expected. This is achieved by selecting the code generator's *Parameter Check* option. An incorrect input parameter causes deadlock or is ignored, depending on the setting of the code generator's *Repeat Behaviour* option.

A new complication with inputs is now evident. Suppose that alternative inputs allow *A* or *B* to go off-hook. The two *OffHook* inputs could refer to either of these subscribers. Improperly drawn CRESS diagrams can thus lead to non-determinism. To combat this, the code generator performs a data flow analysis so that it knows which call parameters are defined at each point in the flow graph. The code generator uses this to detect non-deterministic inputs and outputs.

Branches to input nodes cause yet another complication. The SDL syntax forces the code generator to make a *copy* of the input statement at the point the branch occurs. The code then joins the transition associated with the original input.

Sink nodes assign new values to feature parameters and then branch to the new destination node. However SDL syntax requires input statements *before* the sink bindings. In such a case the code generator delays the bindings until after the input. Fortunately the rules of CRESS permit this to be done. Due to loops in the diagram, several nodes may branch to the same destination. If this is an input, a unique state name needs to be used. Shared inputs are therefore labelled as *destination label.source label*. As an example, node 8 in figure 3 is entered from both node 0 and node 7. The first entry gives rise to the state *TESTR.8.TESTR.0*.

Outputs are much more straightforward, though the data flow analysis is used to make sure that output parameters have defined values. An output statement is labelled (e.g. *POTS.2*) so that a later node may branch back to it. Outputs are followed by any associated assignments.

CRESS expressions map very directly to SDL expressions, with just minor syntactic changes (e.g. '%' becomes **Mod**). Only **Not In** needs explicit support – an infix operator '//' that is added to the definition of the *PowerSet* generator. The value **Time** equates to **Now** in SDL. Guards correspond to SDL decisions. An **Else** guard has an exact counterpart in SDL. If a list of guards does not have an **Else**, one is supplied in the generated SDL. However this **Else** leads to deadlock since its execution is a serious error in the description (the guards are incomplete).

As an indication of the complexity in generating SDL, the code for figure 3 starting at node 2 is shown in figure 6. The numbers in boxes attached to nodes of figure 3 indicate the order in which symbols are visited during graph traversal. Although depth-first search is performed, nodes may be drawn in any position in a diagram and so need not be visited left-to-right. Since there are complex loops, some symbols are visited several times. The number for each symbol visit appears on the right of the code in figure 6. The code will require careful study, but the interested reader will find it illuminating. There is insufficient space here to show the comments produced by the code generator (which would make the reading easier).

4.3 Analysing Features using SDL

The SDL generated from the CRESS diagrams may be simulated using a tool like the SDT Simulator. This allows step-by-step manual analysis of the feature behaviour. However this is a tedious procedure that requires the user to be familiar with SDL. It is therefore preferable to use an automated analysis such as supported by the SDT Validator. The *Exhaustive Exploration* option is the most powerful, but tends to reach internal SDT limits rather quickly. *Random Walk* and *Power Walk* are therefore better for validation. By setting parameters like the search

```
State TESTR.2;                                  1
  Input OnHook(B);                              1
   TESTR.2:                                     1
    Output Update(Busy,B,False);                1
    Decision View(Busy)(B);                     2
     (True):                                    2
      TESTR.7:                                  3
       Output Disconnect(A,B);                  3
       NextState TESTR.8;                       4
     (False):                                   4
    EndDecision;                                4
    Decision Not View(Busy)(B);                17
     (True):                                   17
      Join TESTR.0;                            18
     (False):                                  17
    EndDecision;                               17
    Decision View(Busy)(A);                    19
     (True):                                   19
      TESTR.4:                                 20
       Output LineBusyTone(A);                 20
       NextState TESTR.3.TESTR.4;              22
     (False):                                  19
    EndDecision;                               19
    Decision Not View(Busy)(A);                23
     (True):                                   23
      NextState TESTR.2.TESTR.2;               24
     (False):                                  23
      Stop;                                    23
    EndDecision;                               23
State TESTR.2.TESTR.2;                         24
  Input OnHook(B);                             24
    Join TESTR.2;                              24
State TESTR.3.TESTR.4;                         21
  Input Dial(A,B);                             21
    Join TESTR.3;                              21
  Input OnHook(A);                             22
    Join TESTR.5;                              22
State TESTR.8;                                  4
  Input OffHook(A);                             4
   TESTR.8:                                     4
    Output Update(Busy,A,True);                 4
    Task B_0:=B, B:=A, A:=B_0;                  5
  TESTR.0:                                      6
   Output StartRinging(B,A);                    6
   Output Update(Busy,B,True);                  6
   Output Update(Ringing,B,A,True);             6
   NextState TESTR.8.TESTR.0;                   7
State TESTR.8.TESTR.0;                          7
  Input OffHook(A);                             7
    Join TESTR.8;                               7
  Input Dial(A,B);                              8
    TESTR.3:                                    9
    NextState TESTR.3.TESTR.3;                 10
State TESTR.3.TESTR.3;                         10
  Input Dial(A,B);                             10
    Task B:=A;                                  9
    Join TESTR.3;                               9
  Input OnHook(In_0);                          11
    Decision In_0=B;                           11
     (True):                                   11
      Task A:=B;                               11
      Join POTS.9;                             11
     (False):                                  12
       TESTR.5:                                13
         Output Update(Busy,A,False);          13
         NextState TESTR.3.TESTR.5;            14
    EndDecision;                               15
State TESTR.3.TESTR.5;                         15
  Input Dial(A,B);                             15
    Task B:=A;                                 14
    Join TESTR.3;                              15
  Input OnHook(B);                             16
    Join TESTR.2;                              16
```

Figure 6: SDL generated for Part of Figure 3

depth and checking the percentage of symbol coverage, substantially (or completely) the same effect can be achieved as exhaustive exploration.

What emerges from validation is a list of error reports and an MSC describing the validation undertaken. Error reports deal with situations like deadlocks, implicitly consumed inputs, and input queues growing without bound. They all indicate problems with a feature's description.

A single feature can be evaluated with POTS, whether through simulation or validation. The resulting MSC characterises how the feature behaves. This procedure is repeated for each feature. Now all the features can be combined at once. The MSC for each individual feature is used to validate the composite behaviour. If there is no interaction, the feature will behave exactly as before. If there is interaction, the validation will fail (generally through deadlock). The validator error reports give a trace of the signals leading up to failure. It is then up to the designer to resolve the interaction by changing the description of one (or more) features. As an alternative, a number of the techniques developed for LOTOS [3] can also be applied to SDL. For example observers, watchdogs and analysis of traces have direct counterparts in SDL.

Since SDL is close to a programming language, SDL tools generate code in conventional languages like C. In principle, this code could be embedded directly in a switch. Several companies use compiled SDL in just this way, so CRESS offers an interesting alternative for generating feature code.

5 Supporting CRESS with LOTOS

5.1 LOTOS Specification Framework

The LOTOS framework resembles that for SDL, except that subscribers interact directly with the switch because communication in LOTOS is synchronous. A switch instance synchronises only with the subscribers in a call. The switch process communicates on gate *User* with the subscribers, *Bill* with the billing system, *Stat* with the status manager, and *Scp* with the SCP. The CRESS preprocessor automatically detects which code generator to use – here LOTOS. *Cress(Types)* defines the required data types. *Cress(CFBL,CND,INCF,TWC)* instantiates the top-level switch process, followed by its definition for these diagrams. The '*-n 10*' parameter is an example of a code generator option, here the maximum number of switch process instances.

```
Specification Network [User] : NoExit
 Cress(Types)
  Behaviour
    Hide Bill,Stat,Scp In
      ( (StatusManager [Stat] |[Stat]| SCP [Scp,Stat])  ||| BillingSystem [Bill])
    ||
      Cress(−n 10,CFBL,CND,INCF,TWC)
  Process BillingSystem [Bill] : NoExit := ...
  Process StatusManager [Stat] : NoExit := ...
  Process SCP [Scp,Stat] : NoExit := ...
EndSpec
```

5.2 LOTOS Code Generator

For comparison of LOTOS code generation with SDL, the translation of figure 3 starting at node 2 is shown in figure 7. The LOTOS code generator is structurally similar to that for SDL, but is significantly simpler. This is largely because inputs are completely straightforward in LOTOS. In fact, LOTOS does not distinguish between input and output at all; the use of '?' for input and '!' for output is essentially conventional. The code generator translates CRESS inputs and outputs to LOTOS in exactly the same way. The only slight difference is that CRESS output parameters are checked to have defined values using the data flow analysis. Node 1 of figure 1 is translated to *User !OffHook ?A:Address* because *A* is known to be undefined at this point. However node 2 is translated to *User !DialTone !A* as *A* is now defined.

Assignments associated with input or output are translated after the corresponding event. Feature parameters can be used directly, and are updated in a **Let** statement. Use of a call status variable like *Busy B* requires a prior event like *Stat !Read !Busy !B ?BusyB:Bool* that synchronises with the status manager. Translating an expression is therefore slightly tricky, because all such variables need to be read before the code for the expression is generated. Updating a call status variable also requires synchronisation with the status manager, e.g. *Stat !Write !AudibleRinging !A !B !True*.

CRESS expressions map fairly directly to LOTOS expressions, with just minor syntactic changes (e.g. '<=' becomes *Lt*). Since LOTOS does not have an **if** construct, the translation calls a

```
Process TESTR_2 [Bill,SCP,Stat,User]                    1
 (A,B:Address) : NoExit :=                              1
  User !OnHook ?B:Address;                              1
  Stat !Write !Busy !B !False;                          1
  Stat !Read !Busy !B ?BusyB:Bool;                      2
  Stat !Read !Busy !A ?BusyA:Bool;                     19
  (                                                     2
    [BusyB] ->                                          2
      User !Disconnect !A !B;                           3
      TESTR_8 [Bill,SCP,Stat,User] (A,B)                4
  []                                                   17
    [Not (BusyB)] ->                                   17
      TESTR_0 [Bill,SCP,Stat,User] (A,B)                6
  []                                                   19
    [BusyA] ->                                         19
      User !LineBusyTone !A;                           20
      (                                                21
        TESTR_3 [Bill,SCP,Stat,User] (A,B)             21
      []                                               22
        TESTR_5 [Bill,SCP,Stat,User] (A,B)             22
      )                                                22
  []                                                   23
    [Not (BusyA)] ->                                   23
      TESTR_2 [Bill,SCP,Stat,User] (A,B)               24
  )                                                    24
EndProc                                                24
Process TESTR_8 [Bill,SCP,Stat,User]                    4
 (A,B:Address) : NoExit :=                              4
  User !OffHook !A;                                     4
  Stat !Write !Busy !A !True;                           4
  TESTR_0 [Bill,SCP,Stat,User] (B,A)                    5
EndProc                                                 5
```

```
Process TESTR_0 [Bill,SCP,Stat,User]                    6
 (A,B:Address) : NoExit :=                              6
  User !StartRinging !B !A;                             6
  Stat !Write !Busy !B !True;                           6
  Stat !Write !Ringing !B !A !True;                     6
  (                                                     7
    TESTR_8 [Bill,SCP,Stat,User] (A,B)                  7
  []                                                    8
    TESTR_3 [Bill,SCP,Stat,User] (A,B)                  8
  )                                                     8
EndProc                                                 8
Process TESTR_3 [Bill,SCP,Stat,User]                    8
 (A,B:Address) : NoExit :=                              8
  User !Dial !A !B;                                     8
  (                                                     9
    TESTR_3 [Bill,SCP,Stat,User] (A,A)                  9
  []                                                   11
    POTS_9 [Bill,SCP,Stat,User] (B,B)                  11
  []                                                   12
    TESTR_5 [Bill,SCP,Stat,User] (A,B)                 12
  )                                                    12
EndProc                                                12
Process TESTR_5 [Bill,SCP,Stat,User]                   13
 (A,B:Address) : NoExit :=                             13
  User !OnHook !A;                                     13
  Stat !Write !Busy !A !False;                         13
  (                                                    14
    TESTR_3 [Bill,SCP,Stat,User] (A,A)                 14
  []                                                   16
    TESTR_2 [Bill,SCP,Stat,User] (A,B)                 16
  )                                                    16
EndProc                                                16
```

Figure 7: LOTOS generated for Part of Figure 3

specification-defined operation, for example *Conditional(Time Lt 0900,CheapRate,PeakRate)*. If the value **Time** appears in an expression, it is first read from the status manager's clock: *Stat !Read !Clock ?Time:Time*. CRESS guards equate directly to LOTOS guards. The code generator scans all guards in the list following a node, first reading all the status variables needed and then translating the guards.

The main complication in the LOTOS translation arises where several nodes lead to the same shared node. This requires a process that starts at the shared node. All such processes are parameterised by the standard gates (*Bill, Scp, Stat, User*) and the feature parameters. Processes are named after the node label (e.g. *POTS_1* for node 1 in figure 1). When a shared node is entered, whether directly or via a sink node, the LOTOS translation is a call of the corresponding process. In the case of a sink node, the binding is used to define the call parameters.

5.3 Analysing Features using LOTOS

Features are simulated and analysed much as for SDL. LOLA (LOTOS Laboratory) is very convenient for this. Step-by-step simulation is possible but tedious. Instead the *VarExpand* function of LOLA is used to explore each feature's behaviour to a certain depth. This creates a test process that can be used with LOLA's *TextExpand* function to check if the feature behaves the same way when combined with a number of other features.

In fact it is hardly necessary to develop new techniques for detecting feature interactions using LOTOS. [3] describes techniques that can be used directly with LOTOS generated by CRESS. The only difference is that in the author's case the LOTOS is generated automatically instead of being hand-written. This ensures that the LOTOS reflects the CRESS descriptions faithfully, and simplifies maintenance and extension of the descriptions.

6 Conclusion

As has been described, CRESS has significantly tightened up and extended CHISEL as a rigorous notation for describing services and features. The availability of tools to check CRESS diagrams for static correctness is an important gain. The open, multi-platform nature of these tools makes of them of potential widespread value. The author intends to distribute the tools freely to other organisations for research purposes. By doing so, it is hoped that CHISEL (in its CRESS extension) can become a shared notation within the community for describing features. Efforts towards feature interaction have been diluted as each researcher has needed to develop feature descriptions from scratch. It is hoped that others will be helped by the CRESS feature library, tools for creating and checking new features, and code generators for producing other languages.

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. Faci, L. M. S. Logrippo, and B. Stépien. Structural models for specifying telephone systems. *Computer Networks and ISDN Systems*, 29(4):501–528, Mar. 1997.

[3] Q. Fu, P. Harnois, L. M. S. Logrippo, and J. Sincennes. Feature interaction detection: A LOTOS-based approach. *Computer Networks and ISDN Systems*, 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. *Specification and Description Language*. ITU-T Z.100. International Telecommunications Union, Geneva, Switzerland, 1996.

[7] 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.

[8] F. Lucidi, A. Tosti, and S. Trigila. Object-oriented modelling of advanced IN services with SDL-92. In Z. Brezocnik and T. Kapus, editors, *Proc. COST 247 International Workshop on Applied Formal Methods*, Slovenia, June 1996. University of Maribor.

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

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